ASSIGNMENT 1 – Implementing a iPod/Zen/Zune-style Music player
COMP10050
Spring 2023/2024
Simon Lasak – 23721209

This document provides implementation details and contains information about my decisions in the Music player C source code.

Structure

The project consists of three source files: assignment1_main.c, helpAndPrint.c, and sortAndShuffle.c.

For the latter two header files are provided for clearer implementation into the main code.

assignment1_main.c

The main file is used to declare variables, arrays and run the whole code. It also contains two functions for handling the input from the user.

helpAndPrint.c

This program contains functions needed for printing, counting, copying, and formatting of strings. Most of the functions prepare the variables and strings needed for easier manipulation and to divide the code into separate segments that can be modified more clearly.

The most important for correct shuffling is the function **fillarray()**. Its crucial role is to provide an array filled with each song twice. However, it does in a way that respects the MIN_DISTANCE between songs if at least MIN_DISTANCE+1 songs are provided. This is a key element for shuffling later because we know that the same songs are already at safe distance from each other.

sortAndShuffle.c

Here, the most important functions are kept to sort, shuffle and finally print the shuffled playlist. For sorting I used selection sort. There are two shuffling algorithms included, The Fisher Yates and a custom shuffling algorithm to keep the distance between the identical songs.

Input from the command line

For input I used the function **fgets()** to scan the strings provided. fgets() has the benefit of reading the whole line of input, whereas if I had used scanf() everything after the first white space would be ignored. fgets() lets me save multiple word strings terminated with \0. The function also makes sure the MAX_LENGTH of characters is respected -> helps us to avoid undefined behaviour.

```
fgets(artists[numOfArtists], MAX_LENGTH, stdin);
```

The drawback of fgets() I had to deal with is that it also included the new line character at the end of each string. This lead mostly to output formatting errors.

I used the function **strcspn()** to find the index of the new line character and replace the character at that index with the NULL character -> signalling the end of the string.

```
artists[numOfArtists][strcspn(artists[numOfArtists], "\n") - 1] = 0;
```

The input algorithm also takes care of multiple constraints specified.

- at least one artist/song and up to MAX_ARTISTS/MAX_SONGS,
- if "ENTER" key is pressed the input ends

To make sure that at least one string is provided a do-while loop is used which checks whether the first character of the first string is the new line character.

Also, the program does not let the user provide an Artist without at leas one song for them.

If the user presses enter after at least one artist and their songs, the program breaks out of the function and moves on to sorting. Similarly, after at least one song is provided the user can end the song input with Enter key and input a different artist if we have not reached MAX_ARTISTS.

Sorting algorithm

I used **selection sort** to sort the artists and their songs separately. Selection sort sorts the array in place and does not need any additional auxiliary array. In my code for sorting, I only used two additional variables:

- one string “swap” to temporarily hold the string that is being moved,
- and an integer minIndex that has value i of the outer loop at the beginning of each iteration and changed to j if there is a string with lower ASCII value found by the inner loop

For comparing the two strings I used **strcasecmp** function instead of strcmp function, because strcasecmp is case sensitive and lets us avoid the case where, for instance "Anvil" would have larger Ascii value than "xylophone".

```
            if(strcasecmp(array[j], array[minIndex]) < 0)
            {
                minIndex = j;
            }
```

If array[j] has a lower ascii value than array[minIndex] change the index.

Shuffling algorithm

My shuffling algorithm is a process that begins first by checking the total number of songs provided by the user and deciding whether it is possible to maintain the MIN_DISTANCE of 5 stated by the specification of the assignment. However, this **MIN_DISTANCE can be changed by the user** to whatever number the user wants in the #define MIN_DISTANCE line.

If the user provides **less or equal songs than the MIN_DISTANCE** the distance cannot be achieved. For this case I use the **Fisher-Yates shuffle** which has no constraints. The Fisher-Yates is a simple quick shuffling algorithm that uses just a single for loop and I find it efficient and neat for shuffling an array with no constraints. However, I found it difficult to modify it to respect the MIN_DISTANCE constraints so I used a different approach for this case.

If the user provides more songs than the MIN_DISTANCE the distance between songs is achieved by the **shuffleSongsWithDist()** function. This algorithm works with a single array, one variable string to hold the string being moved and a "boolean" variable to know if the song is in the first MIN_DISTANCE positions of the playlist. For simpler explanation I will refer to the first MIN_DISTANCE elements of the array as the "queue". I am not certain if there is a name for this specific shuffle but in simple terms it works as follows:

- it iterates through the sorted playlist from 0 to the end of the playlist.
- At each iteration, an inner loop generates a random number in range [i, end of the playlist]
- After the random number "j" is generated, the program checks whether the song at position j is also at the first MIN_INDEX positions of the playlist.
- If the song at random index j is not in the queue, the song[j] is copied to the swap string, all of the songs in range [0, j] shift one place to the right, then the random chosen song is put into the first position of the playlist.
- If the song is in the queue, the inqueue variable is set to 1
- Then when the song[j] is in queue the program checks whether it also is at least MIN_INDEX songs away from the end of the queue (ensuring the distance is kept without putting it at the start of the playlist).
- The condition before is only fully true and valuable if the indices j and i are equal. Meaning that the random song chosen is the first song in the unshuffled part of the playlist.
- If all the three conditions for this special case are true we can "skip" this index, leave it in place and can be certain that it is shuffled properly because we know that further down the playlist the same song will not appear again if it was already in queue.

For more detailed information check the comments in the code please.