Institute of Computer Science

CMSC 21: Fundamentals of Programming

Module JOURNAL

Instructions: Accomplish this every week so we can monitor your progress and improve everyone's learning experience. Completely and honestly answer the following questions in relation to this module's lesson/topic.

Submission:
Filename: "<surname><FirstnameInitial><MiddleInitial>_module0#_journal.pdf"
Example: "delaCruzJC_week01_journal.pdf"
Submit to Google Classroom on Module 01

***Accomplish this journal when you are done (or almost done) with the requirements for the module.*

Name:	**Keith Ginoel S. Gabinete**	Module No:	**6**
Student Number:	**2020-03670**	Date:	**08/19/2021**

- Choose at least one of the things discussed that you understood the most. Imagine explaining it to a classmate. **Explain it in your own words**.

I learned that you can create flexible data structures in C that can grow and shrink at runtime by allocating and deallocating memory that can easily accommodate varying data sizes in your C program. These special data structures are commonly known as linked lists. Linked lists are sequences of data structures chained together via links (the same reason they are called linked lists). Many say that linked lists are good alternatives for arrays as they are more efficient in memory consumption and are much easier to manipulate. In fact, it is the second most-used data structure after array <https://www.tutorialspoint.com/data_structures_algorithms/linked_list_program_in_c.htm>. Its concept about linking structures one after the other through structure pointers or arrow operators (->) makes the feeling of working with them more fun and exciting. By usage convention, a linked list is usually composed of structure variables that hold data responsible in creating a chain of information and a 'HEAD' structure pointer that points the first element of the list. Shrinking and expanding lists make use of the functions malloc() and free() defined in the stdlib.h preprocessor directive.

In creating a linked list, one needs to have a pointer to a structure of your desired data type. This pointer is what we call the head pointer. As mentioned earlier, it is a pointer that points to the first element of the non-empty list. If the list is empty, it is essential to initialize the value of head pointer as NULL since it is used to maintain and access the linked list.

One useful tip I learned while dealing with linked lists is that drawing nodes that represent connections of different data structures linked together was surely helpful for beginners like me to visualize and understand the concept behind linked lists with less confusion and complexity.

There are several operations one can perform with linked lists. But since I'm only new to the idea of linked lists as well, I will only discuss the very basic operations I have enough knowledge about. One of these basic operations with linked lists is the addition of data at its beginning (insert at head). Steps are to be followed to easily understand its concept. First, one needs to declare a new structure variable with the malloc() function using a pointer of the same data

type. We can then initialize the members of the structure either by 0, NULL or any other data we can think of but still of the same data type. Secondly, we can point the next pointer of the newly allocated structure variable to the new first element of the list. Lastly, we can then point the original head pointer to the newly allocated structure - thus, inserting a new data between the head pointer and the previous first element of the list (visually imagining about it).

If we do have an operation that involves adding data at head, there's also a way to add data at the end of the list (insert at tail). The first part behind this idea is not far from different than that of inserting data at head - that is, of course, the need to create a new node with malloc() function then initializing its next pointer point to NULL (to indicate last node) . The next part is where it gets a little tricky as you need to make a loop to traverse the linked list. This process is necessary to find the current last node of the list. Note that, since inserting at tail always make use of a next pointer, it is advisable to use inserting at head operation when dealing with empty/NULL head since it cannot have a next structure it can point to; thus, making the program encounter segmentation fault during runtime. One also needs to to create a temporary pointer while traversing through the nodes in the linked list as doing it with the original head might make you lose track of the first element of the list. The last step is to point the the next pointer of the current last node point to the newly allocated node; thus, inserting data at the very end of the list (visually imagining it again).

Just as we can add data at the beginning and at the end of the linked list, we also have operations dealing with deletion of data at its head and at its tail.

For deleting data at the beginning of the list, we allocate a temporary pointer again to point at the current first element of the list. Having the temp pointer points at the first element of the list, we can then just casually move the head pointer to point to the current second data in the list. free() function would then be used to delete the node pointed by the temporary pointer - making the data pointed by the head be the new first element of the list.

On the other hand, deleting data at the end of the list, (just like adding data at tail) requires the use of a loop to traverse through the nodes of the list (not to find the current last node but to find the second to the last node of the list). Again, it is always necessary to make sure that the list contains more than one node as applying this operation at such kind of scenario would result to a segmentation fault. Furthermore, if the head is still empty or has a value of NULL (as to say), it is nice to inform the user that there is no data available to be deleted and that if the list only has one node, it is better to delete that particular data using the delete at head operation. Now, upon locating the second to the last node, we can then deallocate the next node to it using the free() function again. After that, the value of the next pointer of the new last element of the list that previously points to the deleted last element should be updated to NULL.

Another basic operation on linked list is the deletion of all the nodes chained together at it. The very idea about this operation is that we just basically run a code for deleting at head operation over and over again through a loop until the head points to a NULL value.

Finally, one last basic operation I learned about singly linked list is the way to print its contents. The idea on this one is very simple too. We just basically point a temporary pointer of the same data type to the first element of the list - loop it to traverse through nodes, then print all the data each node it passes through contains until the temporary pointer points to NULL.

Before closing this discussion, let me share this knowledge I've acquired about passing linked lists as parameters to functions. The idea behind this is actually very simple. One just need to remember that if you have to modify the linked list, you should pass the it by reference to a function; while, if you just want to print some of its data, search for specific values or edit some data in it or any other actions I haven't mentioned that doesn't modify the linked list, then you can just simply use the pass by value approach.

- What problem/confusion did you encounter about the module lessons? **Explain the problem.**

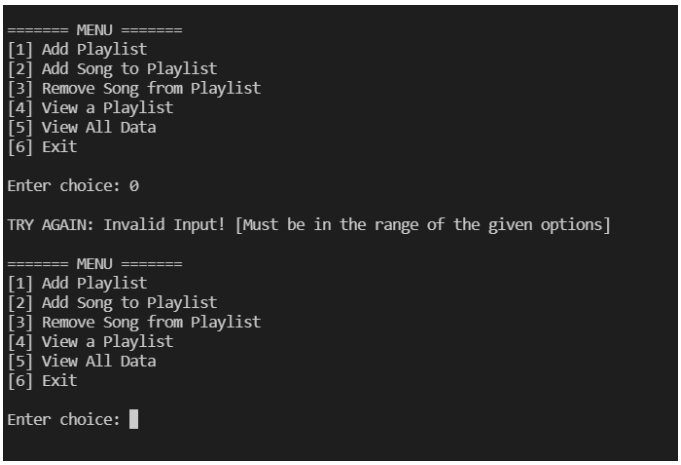
1. As I was planning to convert my program for the individual exercise for module 5 to a program that uses more of linked lists (for exercise in module 6) than just normal structure variables, I found out that there are actually still lots of errors in that code about dealing with input restrictions. Since the exercise in module 5 and module 6 are not so far different from each other (as both programs were intended to store playlists in a file, and that exercise number 5 should be made error-free to move on and work efficiently on exercise for module 6) I would just discuss it here in this journal since dealing with these errors on input restrictions have been the greatest problem I encountered both in exercises for modules 5 and 6.

a. restrictions on range about integer inputs

I already defined a function in my program that returns an integer value that can be used for some selection-based menus. I've used it both for asking a choice from the user (at the main menu) and for asking about the index of certain playlist/song records the user desires to delete or modify

The problem occurred when the user enters a valid integer input in a given range of presented options but is outside of the range on its function definition. Let's say I entered number 0 as an input. Originally in my function definition, 0 as an input should be deemed invalid [as to follow the range given on the main menu (starts at option number 1)] as I do encounter conflicts allowing 0 to be a valid input since the main program is in loop. Since 0 is still not a valid input, but I chose to accept it [since other parts of the program that requires integer input needed it (like asking for an index of the song/playlist)], I left no choice but to have the prompt informing the user that it is an invalid input be written somewhere in the switch statements for valid options in main menu. However, upon doing so would result in the iteration of the printing of the main menu when I don't want it to be.

Problem:



Goal:

```
======= MENU =======
[1] Add Playlist
[2] Add Song to Playlist
[3] Remove Song from Playlist
[4] View a Playlist
[5] View All Data
[6] Exit

Enter choice: 0

TRY AGAIN: Invalid Input! [Must be in the range of the given options]

Enter choice: ^C
```

b. prompts telling the user that the valid data he/she entered would not be stored in a file since the limit of data set by the instructions for the program has already been reached ()

c. comparing two lowercase version of the strings if they are equal or not (checking if the name of the playlist entered by the user already exists)

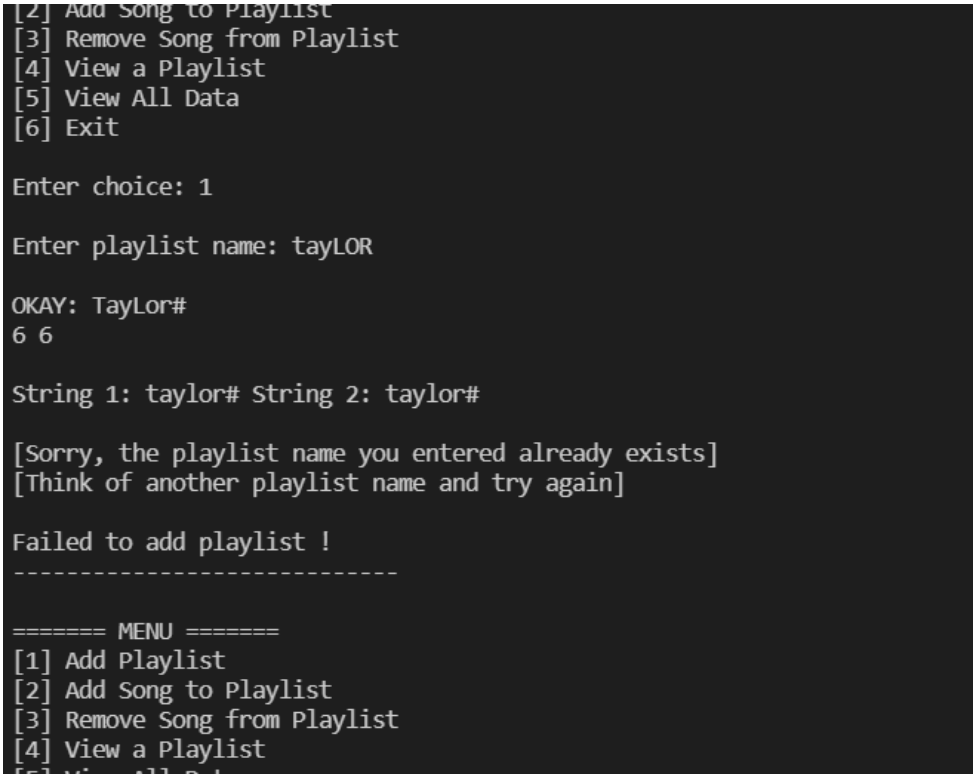
We know that just by using the strcmp() function defined in the string.h library we would be able to easily check if two strings are equal or not. However, this strcmp() function is case sensitive – meaning, it would tell us that strings “DOG” and “dog” are different from each other as ‘d’ and ‘D’ have different ASCII code values. Thus, I created a function that would convert two strings (that would be compared) to their lowercase versions first then compare them normally with the strcmp() function.

Function I defined:

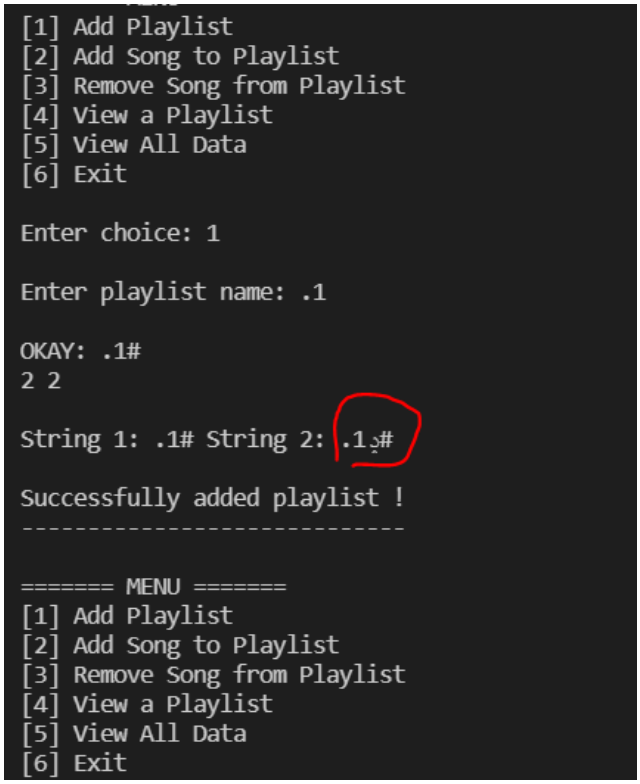
```c
// define a function that compares strings
// return 1 if their lowercase versions are equal
int areEqual (char *string_1, char *string_2, int length_1, int length_2) {
    // declare strings that will temporarily store modified strings
    char temp_1[128], temp_2[128];

    // converts two strings to lowercase letters
    for (int i = 0; string_1[i]!='\0' && i!= length_1; i++) {
        if (isalpha(string_1[i]) != 0) {
            temp_1[i] = tolower(string_1[i]);
        } else {
            temp_1[i] = string_1[i];
        }
    }

    for (int i = 0; string_2[i]!='\0' && i!=length_2; i++) {
        if (isalpha(string_2[i]) != 0) {
            temp_2[i] = tolower(string_2[i]);
        } else {
            temp_2[i] = string_2[i];
        }
    }

    printf("\nString 1: %s# String 2: %s#\n", temp_1, temp_2);

    if (strcmp(temp_1, temp_2) == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

This works as intended if the inputs consist only of alphabet letters. However, the input string already contains characters and numbers, random characters are being thrown to the temp_1 or/and temp_2 variables (variables to be used at strcmp function).

String with alphabets only:



String with numbers/punctuations:



- How did you solve it? **Explain the fix or solution found.**

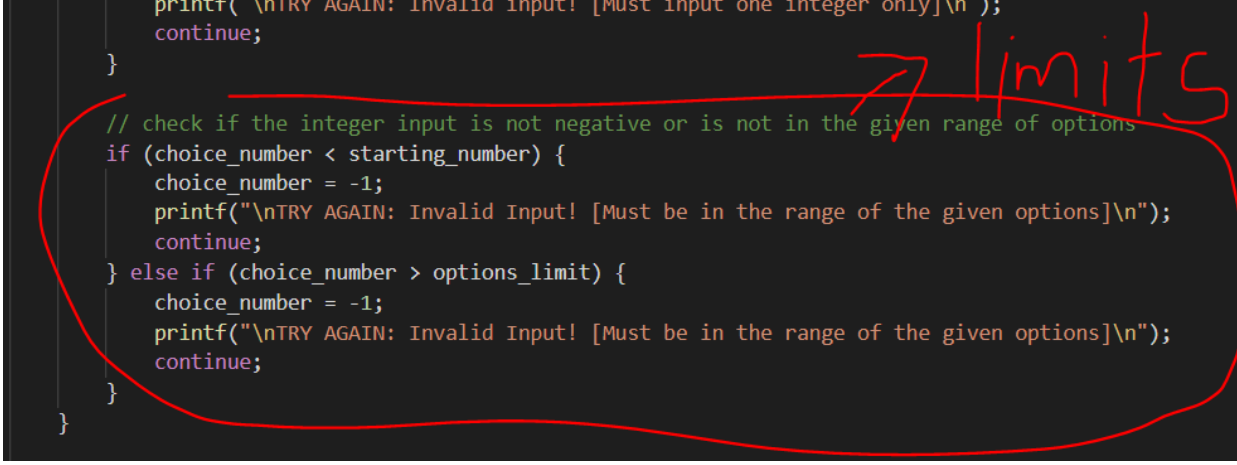
1. a. Upon thinking deeply for an hour, I was able to find out that the solution for this problem of mine is just to add extra parameters to the input function that would set limits to valid input integers:

Original: has no parameter.

```
// define a function that asks for an input integer from the user
// this function will be used in various selection menus
int choiceInput () {
    char fgets_choice[32];
    char extra_char;
    int choice_number = -1;

    while (choice_number == -1) {
```

Modified:

```
// define a function that asks for an input integer from the user
// this function will be used in various selection menus
int choiceInput (int starting_number, int options_limit, char what_to_enter[]) {
    char fgets_choice[32];
    char extra_char;
    int choice_number = -1;

    while (choice_number == -1) {
        printf("\nEnter %s: ", what_to_enter);
        fgets(fgets_choice, 32, stdin);

        // An input is invalid if the user enters a character or if he/she enters more than one integer
        if (sscanf(fgets_choice, " %d %c", &choice_number, &extra_char) != 1) {
            choice_number = -1;
            printf("\nTRY AGAIN: Invalid input! [Must input one integer only]\n");
            continue;
        }

        // check if the integer input is not negative or is not in the given range of options
        if (choice_number < starting_number) {
            choice_number = -1;
            printf("\nTRY AGAIN: Invalid Input! [Must be in the range of the given options]\n");
            continue;
        } else if (choice_number > options_limit) {
            choice_number = -1;
            printf("\nTRY AGAIN: Invalid Input! [Must be in the range of the given options]\n");
            continue;
        }
    }
```



Test run:

```
======= MENU =======
[1] Add Playlist
[2] Add Song to Playlist
[3] Remove Song from Playlist
[4] View a Playlist
[5] View All Data
[6] Exit

Enter choice: 0

TRY AGAIN: Invalid Input! [Must be in the range of the given options]

Enter choice: 0

TRY AGAIN: Invalid Input! [Must be in the range of the given options]

Enter choice: 0

TRY AGAIN: Invalid Input! [Must be in the range of the given options]

Enter choice: 1

Enter playlist name:
```

b. First, I've tried solving this problem on my own for several hours. Having a realization that I couldn't solve it on my own I then browsed through the internet for possible solutions. To my surprise the solution for this is just so simple that I couldn't stop myself from laughing of how stupid I am not to know about this stuff. Thankfully I know it now. I learned from this forum in stackoverflow <https://stackoverflow.com/questions/16953115/why-do-i-keep-getting-extra-characters-at-the-end-of-my-string> that looping through a string to convert it to its lowercase version and storing the results in some string variable, would not actually automatically append the NULL character at the end. Thus, to solve this issue one just need to manually put the NULL character at the end of the resulting string.

Original code:

```c
// define a function that compares strings
// return 1 if their lowercase versions are equal
int areEqual (char *string_1, char *string_2, int length_1, int length_2) {
    // declare strings that will temporarily store modified strings
    char temp_1[128], temp_2[128];

    // converts two strings to lowercase letters
    for (int i = 0; string_1[i]!='\0' && i!= length_1; i++) {
        if (isalpha(string_1[i]) != 0) {
            temp_1[i] = tolower(string_1[i]);
        } else {
            temp_1[i] = string_1[i];
        }
    }

    for (int i = 0; string_2[i]!='\0' && i!=length_2; i++) {
        if (isalpha(string_2[i]) != 0) {
            temp_2[i] = tolower(string_2[i]);
        } else {
            temp_2[i] = string_2[i];
        }
    }

    printf("\nString 1: %s# String 2: %s#\n", temp_1, temp_2);

    if (strcmp(temp_1, temp_2) == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

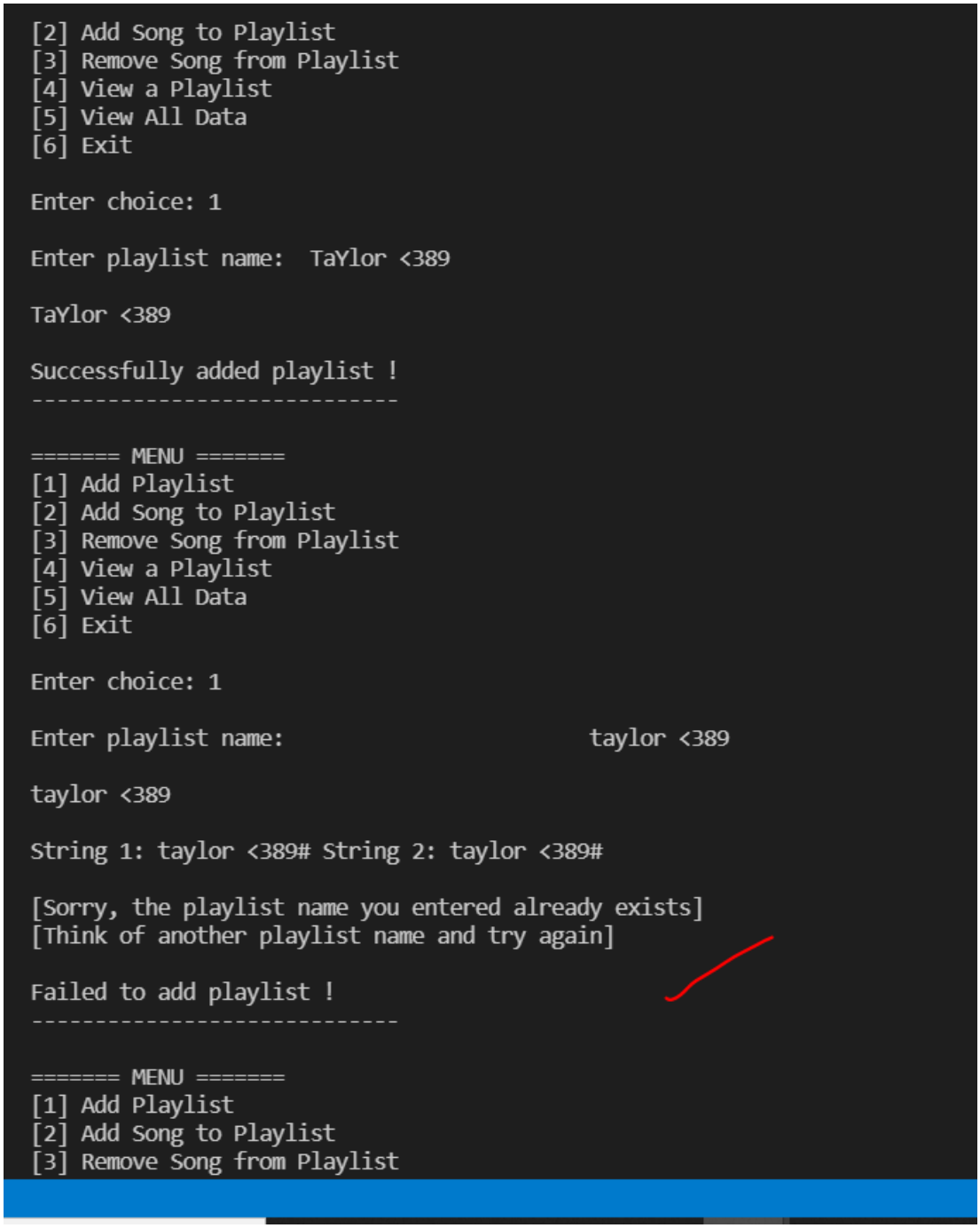
Test run:

```
[1] Add Playlist
[2] Add Song to Playlist
[3] Remove Song from Playlist
[4] View a Playlist
[5] View All Data
[6] Exit

Enter choice: 1

Enter playlist name: .1

OKAY: .1#
2 2

String 1: .1# String 2: .1#

Successfully added playlist !
----------------------------
```

Solution:

```
// define a function that compares strings
// return 1 if their lowercase versions are equal
int areEqual (char *string_1, char *string_2, int length_1, int length_2) {
    // declare strings that will temporarily store modified strings
    char temp_1[128], temp_2[128];
    int i, j;

    // converts two strings to lowercase letters
    for (int i = 0; i<length_1; i++) {
        if (isalpha(string_1[i]) != 0) {
            temp_1[i] = tolower(string_1[i]);
        } else {
            temp_1[i] = string_1[i];
        }
    }
    temp_1[length_1] = '\0';

    for (j = 0; j<length_2; j++) {
        if (isalpha(string_2[j]) != 0) {
            temp_2[j] = tolower(string_2[j]);
        } else {
            temp_2[j] = string_2[j];
        }
    }
    temp_2[length_2] = '\0';

    printf("\nString 1: %s# String 2: %s#\n", temp_1, temp_2);

    if (strcmp(temp_1, temp_2) == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Test run:



- Comments/Suggestions (Optional but we will appreciate it if you tell us one you have in mind)