Group 3- FlashCards

Anto John Mathews- 13205479

Ryan Lawton - 12545341

Ayush Bhatia - 12604486

Jenny Tran - 13244571

Gordon Wang - 13225964

48430 Fundamentals of C Programming

University of Technology, Sydney

2019

Table of Contents

1. Objective

The objective of this project is to develop a terminal based C program that simulates flashcards. This program must allow users to create personalized flashcards which are stored in the user's personalized decks and also must allow users to access and utilize community generates decks. These decks must be playable for the user who must possess the ability to view the question randomly selected from the deck and be allowed to guess the solution. Once the user attempts all questions on the flashcards in a deck the program must generate the performance statistics for the user on a per deck basis. The user's statical data generated must be stored. Also, this program must allow the saving of the flashcards into named and order decks as well as organize the storage of deck allowing the user to navigate the decks in a file browser. In order to save these personalized statistics and decks, users will possess the ability to create a personalized account which saves each user specific deck statistics and decks.

2. Scope

2.1 - Project Scope Description

The targeted aim of the project is to create a terminal based flashcards C program with three primary modes

- The generation of Flashcards into organized decks
- Utilization of the stored flashcards: allow the user to answer the Flashcards and display if the user input matches the answer of the flashcard
- Generate performance statistics for each deck played

Apart from these primary modes, the program has multiple secondary modes implemented to increase the complexity and create a more interactive program. These secondary modes are:

- Saving of the flashcards into named decks
- Storing the performance statistics of the deck
- Create an easily navigable menu to utilize the features.

Both the primary and secondary modes are scheduled to be completed within the 5 week time period as outlined within section 2.2

2.2 - Schedule

Week 1

- Identify the appropriate problem that manipulates real-life data involving data compression or encryption (Flashcards)
- Create header and code files (.h/.c) with containing the potential function prototypes and block comments
- Produce the project report with project objective and scope completed

Week 2

- Implement most of the functions and algorithms
- Begin the integration of the functions and algorithms
- Initialize the debugging process to eliminate any bugs within the functions and bugs created through the integration of the functions/ algorithms.
- Record the bugs and issues encountered and alternative pathways as well as how they were found and overcome

Week 3

- Implement all of the functions and algorithms
- Continue the integration of the functions and algorithms
- Continue the debugging process to eliminate any bugs within the functions and bugs created through the integration of the functions/ algorithms.
- Record the bugs and issues encountered and alternative pathways as well as how they were found and overcome

- Continue adding to more sections of the project group (Features and Issues)

Week 4

- Finish the implementation and integration of all the functions and algorithms.
- Continue the debugging process and recording the bugs
- Create a fully functional Flashcards C program
- Finish all sections of the project report
- Begin Presentation Preparation
- Implement final changes to the program and report

Week 5

- Project Presentation

3. Features

3.1 Login Screen

The features allow the consumer to access the decks and stats within the database under the specific login the consumers enter.

3.2 Create Account

The Create Account feature was mainly implemented to ensure that the user's data are private (private decks) and not accessible to others. This feature ensures the program only allows the user to only access the database within the account. This feature also allows for the performance statistics and user decks to be assigned to that specific login. This feature is also vital as the program can only be accessed through the creation of the account. Furthermore, this features allows us to achieve the primary and secondary modes listed in section 2.1.

3.4 Print Menu

The feature is vital to C program as this allows the user to navigate through the other program features.

3.5 Create Deck

This a primary feature of the program to allows creating personalized custom decks. Hence the create deck feature allows the user to create and name their own flashcard deck, allows users to add Flashcards to a deck. Furthermore, the user is allowed to choose to keep their deck private or public.

3.6 View Decks

Allows user to view the decks that they have created and saved from the community deck. This feature achieves the secondary mode of the program of organizing the decks allowing it to be navigated in a file browser referred to in section 2.1.

3.7 View Community Deck

In addition, to create your own private decks, users have the ability to browse to community decks for the user to attempt to learn new topics. Furthermore, the feature allows users to save community decks into their 'deck' to attempt to test their knowledge of the topic within the community. Furthermore, the features allow the users to view the decks they chose to make public and allow other users to add into the deck database.

3.8 View Stat

One of the primary modes is to generate performance statistics for the decks the user plays. The implementation of this feature allows customers to view their progression and regression for the decks hence giving the customer data to improve their knowledge of the topic within the deck. Ultimately, achieving the aim of flashcards by quizzing the individual and aid in remembering the key points of the subject.

3.9 Edit Deck

This feature allows users to alter the flashcards within their deck database. However, the user can only alter their own private decks and cannot alter the community decks. Through this feature, the user can edit any error that occurred in the creation of the flashcards and decks.

3.10 Delete Deck

This feature allows the user to remove the finished or unused decks from the user's deck database

3.11 Play Deck

The features allow the user to play the decks within their collection. The user is shown a random flashcards for the deck and allowed the chance to answer this question.

4. Design & Structure Analysis

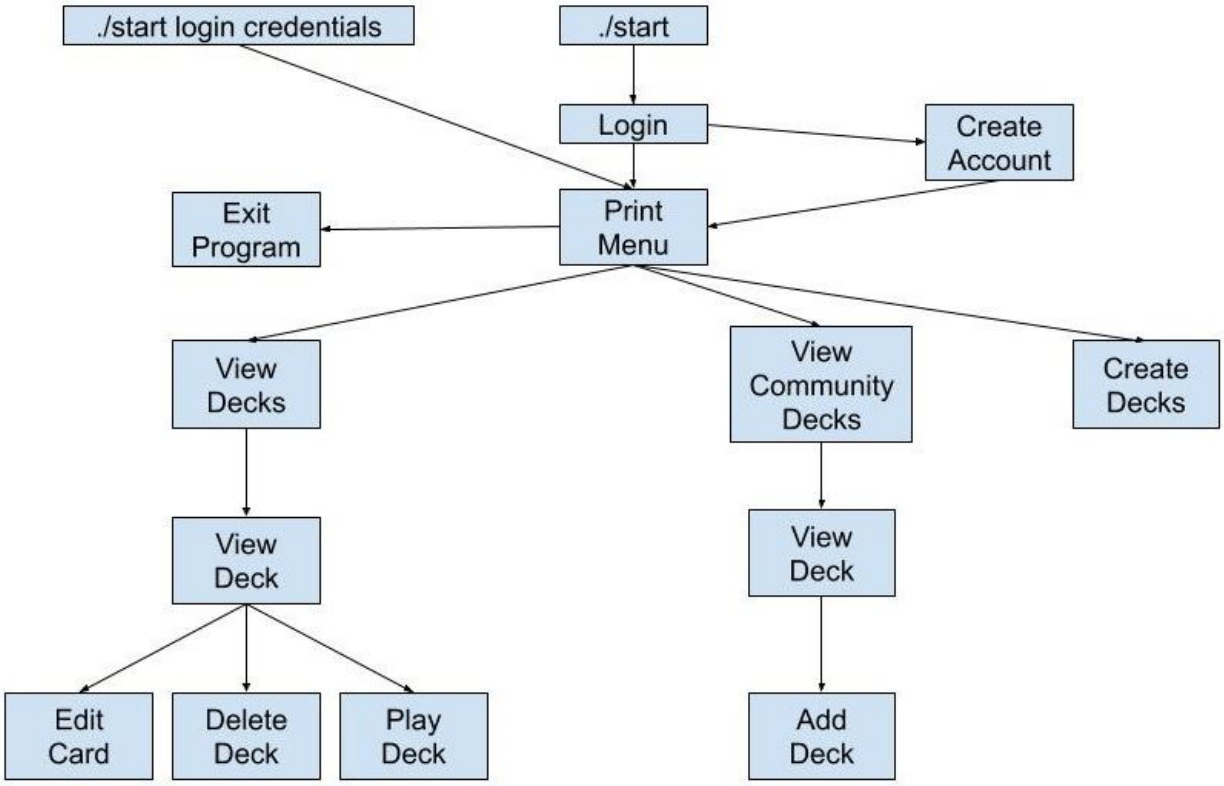

Figure 1: Data flow diagram on how Flashcard program functions.

4.1 Encryption and Compression

At the start of our program, the user will be prompted to login. If the user does not currently have an account, the user can create an account at this stage. The user will be asked to enter a username, display name and password. The username has to be unique and will be compared with other users in the db to verify that it has not been taken. The requirements we set for our password is to have at least one capital letter, one lowercase letter and one number. This information is then compressed using Run Time Encoding, encrypted using an XOR cypher and then stored stored within a DB.

4.2 Choice

The use of switch statements throughout our program made for a better and more intuitive way to navigate our application, though in a few circumstances we opted for string comparison as we thought it felt more natural when choosing which deck to

interact with.

4.3 Login System

The reasoning behind the login system is to allow for the separation of decks and users so that we could store statistics for each user. This is also why we implemented the public deck system, or community decks so that users can add a deck to their list that someone else has made and have their own individual stats for each deck.

4.4 Linked List Implementation

Our decks have many components to them, as such we decided to implement our design structure using a linked list system. Each deck would become a link in a chain, and at each link we would store information like the name of the deck, who the owner is, who the author is, and other statistical information. Being a linked list this also means we store where the next point in the chain is, otherwise we would not know where each element is in memory. A common limitation of linked lists is that it causes you to have to iterate over the entire chain if you want an element, you can't just give it an index. In the future something more akin to a hash map that still allowed iteration would better optimize our program structure. However for the scale of our project a simple linked list was enough to get what we needed to create a functioning application. This feature was not limited to our decks as we also implemented it with the cards as well. The deck would then only have to store the top of the card heap and when needed we could iterate over each card.

4.5 Colour & Screen Navigation

Another design aspect we added was colour so that users could better differentiate between errors, general information and options that the user can take. To better help the user navigate we also implemented a 'clear screen' function so that we could reset the console cursor to the top left of the screen and print all relevant information their. This way users won't have to look for where they are in a program and can intuitively follow it with their eyes.

4.6 Debug Mode

We have also implemented a debug mode for our program which can be enabled at the start of operation by adding debug as an argument in the console './main debug'. This feature was made so that during development we could keep an eye on certain parameters to help with finding bugs in the code. We mainly used this feature to keep an eye on how large our decks were getting to make sure we weren't breaking any features,

and if we noticed any irregularities say after adding a deck we could then backtrack to find the issue.

4.7 XOR Encryption

XOR encryption is an encryption method utilized to encrypt data and is difficult to crack due to the fact that if you XOR two unknown variables the individual is unable to distinguish what the output of the variable will be. In simple terms XOR encryption generates a random encryption key or a fixed encryption key which is utilized to decrypt the encryption by performing the XOR operation again with the key.

This encryption is done by first defining the XOR encryption key and the performing XOR operation of the characters in the string to be encrypted.

We utilized the XOR encryption to hide the account details of the user within the database. The utilization of XOR encryption within our program is demonstrated in table 1 and figure 2 below.

Username	Lawyer
Password	Password1
Full Name	Ryan lawton

Table 1: The user account details before XOR encryption

Figure 2: The user account details after XOR encryption

For the future to have a more secure password encryption system, since this will be utilized for storing our user data, it would be best to encrypt them with something like an AES encryption that has come to be the standard for modern day encryption.

4.8 Run-Length Encoding

Runtime compression or Run-length encoding is a form of lossless data compression in which runs of data are stored as a single data value and count, rather than as the original run. The example of the runtime compression is demonstrated in figure 1 below.

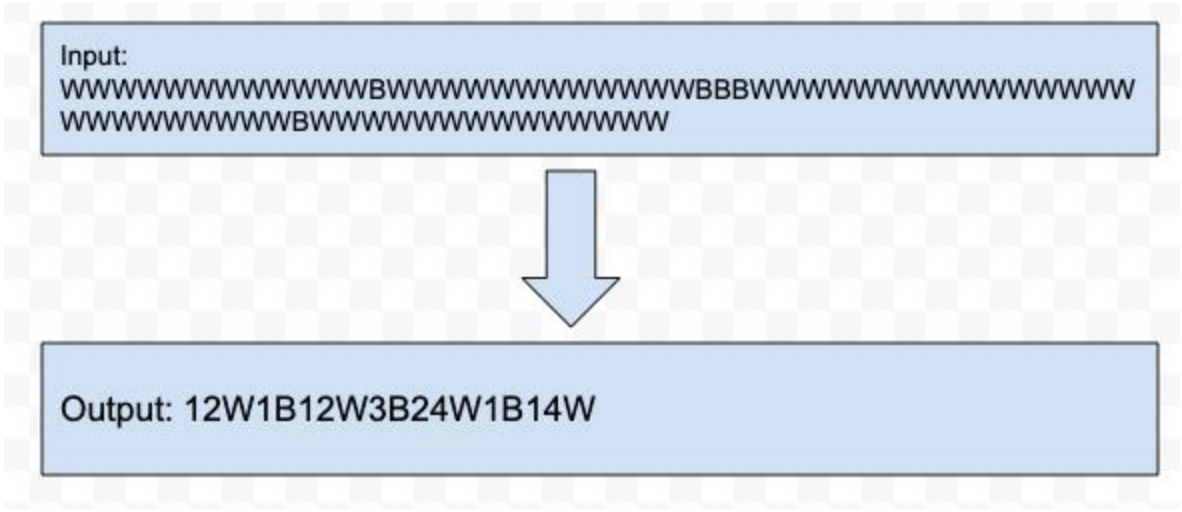

Figure 3: Inputs value before and Output values after runtime compression.

Runtime Compression was utilised within our program in order to reduce the size of the input. Hence for example the “aaaaa” within the program would be reduced to “a5”. However, this was effectively had the opposite impact as the size of the input as expanded. For example the word ‘hello’ would be stored as the ‘h1e1l2o1’.

We decided to use run time encoding originally because it was one of the easiest compression methods to implement, however as we came to realize that this was not the best method for our use case and in practice would make the compressed text larger than before. In the future it would be best to change our compression method to something like a Huffman Encoder that would take advantage of our data types.

5. Issues

5.1 Program Looping

Program infinitely loops, constantly writing to console.

Figure 4: Console stuck, can't interface with it.

To fix this issue we decided to make better use of another issue we found if the wrong input was detected here:

```
default:
    print_red("Invalid choice\n", 1);
    break;
```

Figure 5: Old code before change

What we noticed is there was no break after printing out the error, this caused the input to roll back to printing out the menu UI again, however the input we ask for is of %d using scanf. Due to the limitations of scanf, if there is input that isn't needed, it will be left over in the buffer. The wait function that we have implemented uses a while loop to get each char that may be left in the buffer in order to clear it ready for new input.

```
default:
    clear_screen();
    print_red("Invalid choice\n", 1);
    wait();
    break;
```

```
void wait(){
    char input[100];
    printf("Enter any key\n");
    while((getchar()) != '\n');
    scanf("%[^\n]", input);
    while((getchar()) != '\n');
}
```

Figure 6&7: 7 is the new code, 6 is our wait function.

An alternative way we could have overcome this problem is to change how we approach gathering input. Rather than using scanf, we could fgets and go through it character individually. This way we could better catch invalid input as well as prevent infinite loops. However this approach would require very detailed input checks to verify valid input and for correct strings.

5.2 Memory Leak

```
root@Bahamut:/mnt/c/users/lawyer/documents/github/backup# ./main login Lawyer Password1
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

Figure 8: Console error when encountering the memory leak.

When we were writing our user input system, we used a macro to define the maximum character size that a user can enter. However at the time we had fairly low numbers for a users name, question etc. as we had not yet defined what we would set those values to be, so we didn't run into an issue until when we changed those numbers to be much larger.

```
#define MAX_INPUT_LENGTH 10
```

```
char input[MAX_INPUT_LENGTH];
```

Figure 9&10: 9 is how we defined it in the beginning, 10 is how we set the size of the array.

When we noticed this we defined our maximum input to be large enough for all use cases, and then check for afterwards if the input was too large for the final data type before trying to copy it to that address.

```
#define MAX_INPUT_LENGTH 250
```

Figure 11: New definition that gives some leeway for all our data structures inputs.

Another way we could have fixed this issue is to malloc the memory by using the max size of the destination char address. This way we don't over allocate unnecessary memory while also making sure that if we change the size values of questions or answer in the future, it wouldn't affect the input as the size would then dynamically adjust as well.

6. REFERENCES

- CodingUnit Programming Tutorials 2019, '*Exclusive-OR (XOR) Encryption*', viewed 23 May 2019, <https://www.codingunit.com/exclusive-or-xor-encryption>

- Laing C 2017, *'Cryptography 101 - - XOR Cipher'*, Youtube, 13 August, viewed 23 May 2019, <https://www.youtube.com/watch?v=xK_SqWG9w-Y>

- Lynch W 2019, '*Data Flow Diagram Comprehensive Guide with Examples*' viewed 20 May 2019, < https://medium.com/@warren2lynch/data-flow-diagram-comprehensive-guide-with-examples-d9858387f25e>

- Structured analysis 2019, Wikipedia, viewed 20 May 2019, <https://en.wikipedia.org/wiki/Structured_analysis>

- Run-length encoding 2019, Wikipedia, viewed 22 May 2019, <https://en.wikipedia.org/wiki/Run-length_encoding>