Fundamental of C Programming [48430]

Assignment 3: Group 03

*Sean Simon - 1294568*

*Alexander Leydon - 12912857*

*Joyce Tin Yan Man - 12499947*

*Charles Jean Marc Moreau - 13689068*

*Clement Max Rene Valot - 13697394*

# **Table of Contents**

[**Project Objectives**](https://docs.google.com/document/d/1_yj_8lMEhFLQg3pmyKyfHDhbNpNJqwypGxVsPzQPDmY/edit#heading=h.k81ugyk30klp) **5**

[**Project Scope**](https://docs.google.com/document/d/1_yj_8lMEhFLQg3pmyKyfHDhbNpNJqwypGxVsPzQPDmY/edit#heading=h.c47ghzxc6gp8) **5**

**Illustration 6**

[**Design Discussion**](https://docs.google.com/document/d/1_yj_8lMEhFLQg3pmyKyfHDhbNpNJqwypGxVsPzQPDmY/edit#heading=h.ttdzhchg20aa) **7**

[**Critical Thinking**](https://docs.google.com/document/d/1_yj_8lMEhFLQg3pmyKyfHDhbNpNJqwypGxVsPzQPDmY/edit#heading=h.bxs96ms1xa8e) **10**

[**Appendix**](https://docs.google.com/document/d/1_yj_8lMEhFLQg3pmyKyfHDhbNpNJqwypGxVsPzQPDmY/edit#heading=h.pi3ya1vu69tc) **12**

# **Project Objectives**

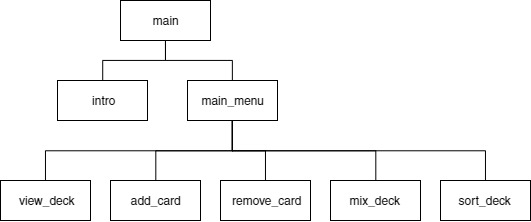
In this project, we aim to create a tool to manage the creation and ordering of standard playing cards. *Pocket Cards* gives users the ability to create their own deck of cards, or choose from a preset. It allows them to add cards to their deck or remove cards. They are also able to sort and shuffle the deck, as well as saving it and loading it in a secure way using encryption and compression methods. This tool would act as a program running on a backend web server for online gaming companies.

# **Project Scope**

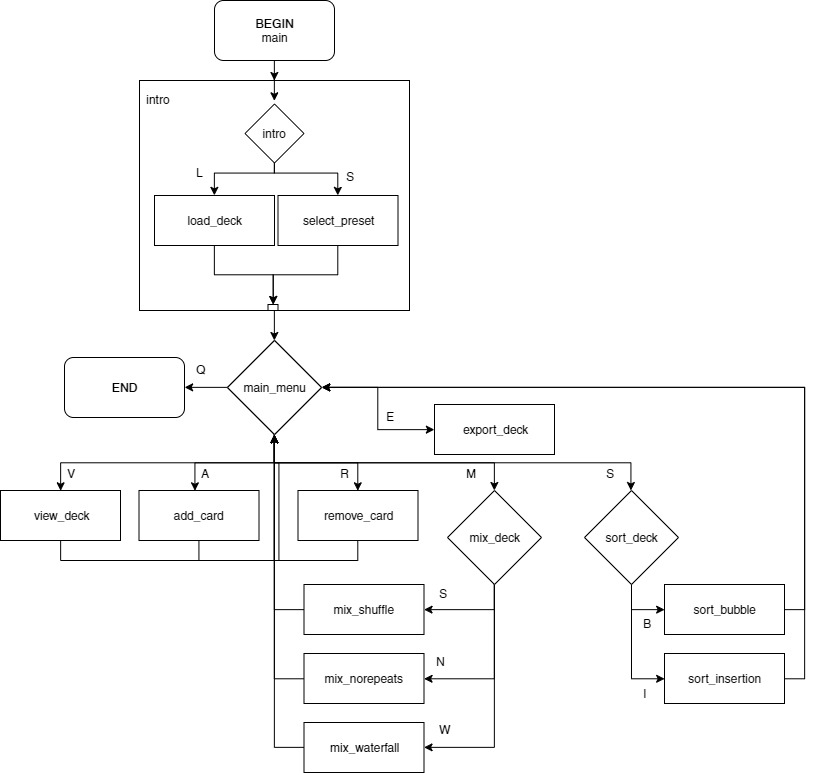
|  |  |  |
| --- | --- | --- |
| **Functionality** | **Priority** | **Completion** |
| Can create a new deck or load an existing one | High | X |
| Can add cards to deck | High | X |
| Can remove cards from deck | High | X |
| Can view deck | High | X |
| Can sort deck | Medium | X |
| Can shuffle the deck | Medium | X |
| Can encrypt cards | Medium | X |
| Can compress and save deck | Medium | X |

**Illustration**

**Structure Chart:**



**Flow Chart:**



# **Design Discussion**

**How cards are stored:**

Cards are stored as their own custom struct, card\_t. Cards have 2 variables, a char for the suite and a char for the value. This was seen as the easiest way to handle them for when it comes to storing, sorting and saving.

**Input Validation:**

The card\_t struct does not have limitations of what the value of its suite and value are. This is instead handled by the input validation. This means that if we wanted to increase the types of cards we could have, such as including the Jester card, we modify the input validation to accept that new card suite/value instead of on the card itself. (EXPAND)

**How collections of cards are stored:**

Collections of cards, such as the deck, are stored internally as an array of card\_t structs rather that a linked list. This is because:

1. The maximum deck size is known: 52
2. Makes accessing quicker as you don’t have to go through the whole list
3. Less likely to cause memory leaks when removing/adding cards from those collections

**Encryption methods used:**

Simple substitution cipher:

The simple substitution cipher takes in input (KEY) one single character. We use the decimal value from the ASCII table of this character to encode the chain as following:

1. Take each character of the chain one by one
2. Take their decimal value from the ASCII table
3. Add the value of the KEY to its value with this formula : 32+(chaine[i]+key)%95; 95 is the number of character that can be printed in the ASCII table and 32 is the ones that can’t.
4. Put this new value in a new chain and return this chain.

To decode an encoded chain we do the same process but with this other formula : 32+(31+chaine[i]-key)%95; 31 being an integer found after several trials of the algorithm.

The problem with this cipher is that it is easy to crack. All we did was to shift all characters of the same amount. A computer can solve it by testing all 95 possible different KEYS that exists (because 0 is the same KEY as 95) and check the result.

Poly Alphanumeric cipher:

The Poly Alphanumeric cipher solves the problem of simplicity of the substitution cipher. Indeed, there are no two different KEYS that can give the same result for the same chain which means there is an infinite number of KEYS.

The Poly Alphanumeric cipher takes in input (KEY) a string (chain of characters). We use the decimal values from the ASCII table of these characters to encode the chain as following:

1. Take each character of the chain one by one
2. Take their decimal value from the ASCII table
3. Add the value of the j character of the KEY to the i+j value of the chain with this formula :

32+(chaine[i+j]+key[j])%127 if chaine[i+j]+key[j]>127

chaine2[i+j]= (chaine[i+j]+key[j])%127 otherwise

Once we reach the end of the KEY ( its last character), we start again to the beginning of the KEY (its first character) until we reach the end of the chain to code.

1. Put this new value in a new chain and return this chain.

To decode an encoded chain we do the same process but with this other formula :

127+(chaine[i+j]-32-key[j]) if chaine[i+j]-key[j]<32; 127 being the total number of elements in the ASCII table and 32 the number of unprintable elements

chaine2[i+j]= (chaine[i+j]-key[j]) otherwise

The most difficult part in this algorithm was to find the special integers to add and subtract in the formulas because it took a lot of trials and unsuccessful attempts.

* Add comparison table of encryption/compression (advantages/disadvantages)

**Compression used:**

What I tried to do in this algorithm is a variant of the Huffman coding. In the real Huffman coding, when there are a lot of different characters, it is almost impossible to have a character binary-coded by 0 or 1. And I told myself it is a shame that we had to skip those extremely small binary code. So I came up with an idea: "What if we sorted all the characters of the chain in a decreasing order in an array and then we give each character the binary code corresponding to its position in the array". Meaning that the most frequent character in the chain, in the first position of the array, would have the binary code matching its position (0) so 0; the second (1) so 1; the third (2) so 10 etc...

It is not really the Huffman code but it does the same thing by matching high frequencies character to low binary code.

**Sorting methods used:**

Bubble Sort:

Bubble sort was known as one of the easiest sorting algorithms among all the sorting algorithms that compare each pair of adjacent items in the array and swaps them if they are not in order. Although it was also one of the slowest algorithms, it takes the least amount of working memory and it's also simple and easy to read by the developer. Besides, the disadvantage of slow processing time will not be significant as the maximum amount of items in the array is only 52.

Insertion Sort:

Insertion sort is adopted to the project because the value of cards is always smaller than or equal to 13 if sort deck by value and not larger than 4 if sort deck by suite, which makes insertion sort an efficient choice among all the sorting algorithms. Insertion sort is also known as a stable and adaptive algorithm as it does not change the relative order of elements with equal keys and efficient for data sets that are already substantially sorted.

Quick Sort:

Quicksort is regarded as the best sorting algorithm. This is because of its significant advantage in terms of efficiency. Besides, quicksort doesn’t require extra storage as it’s sort in place. It quicksort works on the divide-and-conquer principle and the partitioning and arranging process is performed repeatedly on the resulting sublists until the whole list of items is sorted.

**Mixing methods used:**

Twister:

The twister mix swaps 2 cards from the two half of the deck every two cards. I.e. card 2 will swap with card 27.

No repeats:

The no repeat mixing methods mixes the deck in a way that each card can’t be preceded or followed by a card of the same suite or value.

Shuffle:

The shuffle mix randomly mixes the deck of cards.

# **Critical Thinking**

This project presented many problems that intensive critical thinking to overcome.

* As the suites were too big to be stored as chars, and string literals were incompatible with our encryption methods, we had to think of a way to store them. Instead of directly storing the string value for the suite (e.g. ♥), the back-end stores the suite as a char respective to its prefix (e.g. h = heart) and when viewing the cards the program converted the char suite to the corresponding string literal.
* One of the main issues that occurred when loading cards from a file is that it would repeatedly query the user for the filename. If the file did not exist, it would result in an error and ask the user again. This is problematic as a situation would arise where the player had never saved their cards but they would be trapped in this loop. The solution involved using a counter. After several inputs resulting in no file found, the user would be taken back to the main menu.
* In terms of program usability, navigating the menus would be difficult if each menu button was given a number assigned (e.g. [1] View deck, [2] Add a card, ..). This was solved by including the letter of the menu into the option (e.g. [V]iew deck, [A]dd a card).
* Users should not be able to add a card that is already in the deck. This wasn’t working. After some debugging, we found it was due to the input not converting the value and suite were capital chars whilst the cards already in the deck were of lower case chars. We decided that immediately after reading the input, we would convert the input to lowercase so that our checking functions would work.
* Clement (in regards to the menu system) - I chose here to put a do-while loop for one specific reason : we have to do the action (asking the user to enter a value for choice) before checking the condition which is the integer entered by the user must be either S, H, C or A. If not, we go back to the beginning of the do loop, displaying the error message. Each time we start this while loop again, we set the value of invalid to 0 so that it doesn't display the message "Invalid choice" directly while the user didn't even enter a value yet. However, if he enters a wrong choice, invalid would have been incremented before and therefore not be equals to 0 anymore what would display the message.
* Occasionally some comments would cause errors. This occurred when the comment uses a ‘//’ instead of a ‘/\* \*/’ on the university's computers. We went through our code and replaced ‘//’ with ‘/\* \*/’ and where appropriate added individual comment lines into a single block.
* Had issue reading user input after a new line char “\n” as it would save that instead of the actual input. After struggling with it for a while we eventually learnt that placing a spacebar at the start of a scanf will stop this issue. We also tried not using the “\n” char in print statements but that was dependent on the terminal size and terminal font size so we did not use this method.
* Swapping - As sorting functionality and mixing functionality include the concept of ‘swapping’ and the algorithms of swapping will be reused for many times, so we decided to separate it into an identical function. One called ‘swap’ for swapping the memory address of two cards while another one called ‘sawp\_int’ for swapping only the memory address of two integers.
* Swapping - the algorithms that can found online only provide the solution of swapping integer or character but not a structure item. I (Joyce) have applied the knowledge of pointers and address that I learn in class to the method and modify it into a method of swapping a structure object.
* Sorting (Insertion) - Had an issue on swapping cards. After double-checking on the code, it was found that the algorithms could only do the assigning of suite or value to a card but not the swapping cards while performing ‘key condition check’. To solve this issue, I’ve added an extra line to perform the card swapping function.
* Action input validation check - After performing some action to the cards, system will ask for further action within “View / Back / Remove /Add”. For sort and mix function, only View/ back option for the user while adding or remove a card by value or suite function will have View, Back and Remove option for user. Lastly, Remove the specific card from deck function will provide all 4 options. To reduce the redundancy, I create a single function that performs these input validation check by looping through “V/B/R/A” and require a number representing the number of option that system needed when this validation check function being called.
* Compression/Decompression - Had an issue to decompress and read the compressed file because after the compression function, all the different arrays used to write in the file no longer exists after the end of the function. Then to save those values, we had to put the values of the arrays in other file and then in the decompression function read those files in order to have the arrays filled with the good values to read the file and decompress the chain.

# **Appendix**