CS2002-C2-Report

170011474 March 13th

Overview

This practical requires students to implement faro shuffle with in-shuffle and out-shuffle to achieve a magic game. This implementation can be RANKSUIT-oriented or NUMERICAL-oriented and deck contents after every shuffle must be printed out in the terminal.

Design & Implementation

There are 3 source files and 3 header files inside the main practical.

Faro_shuffle.c (main function)

Deck size and arguments from users will be checked. If invalid input appears, proper warnings will be prompted, and the program terminates appropriately. Critical and error-prone parts of implementation are demonstrated below:

1. *Data structures to store a deck of cards: Use LinkedList to store cards (every card is a struct)*. It is the dynamic / heap memory implementations avoid array-based structures entirely. Every card has string to store the content and two cards referring to the previous card and the next card. *In this way, manipulation of shuffling can be straightforward and suits for human's logical thinking. Additionally, inserting a new card can be quick and safe.*
2. *Design of reading input:* input from users will be firstly delivered to an "input" string and the string will be separated into several tokens containing values of cards, such as "AC". strtok() is adopted and the delimiter is the space " ". Every token will be used to make a new Card added to the current decklist.

```
read_line(input, temp_size);

token = strtok(input, delimiter);
while (token != NULL)
{
  addCard(cards, token);
  token = strtok(NULL, delimiter);
}
```

3. *How does it read inputs into the String input?* It assumes the initial size of memory occupied by input cards, and if real contents need more space, then realloc() function will expand the existing memory space to fit the actual size. *The benefits here: can*

maximum the efficiency to use continuous memory blocks without wasting space.

```
while ((c = getchar()) != EOF)
{
  if (c == '\n')
    break;
  if (index == size - 1) // indicates that the current memory
  {
    input = (char *)realloc(input, (size + 1) * sizeof(char));
    size += 1;
  }
  input[index] = c;
  index++;
}
input[index] = '\0'; // append the ending
```

4. *The difficulty in this part is how to pass values of cards without touching the segmentation fault.* Segmentation fault always occurs when people are trying to access read-only memory. This bug can be dealt with proper initialisation and malloc(), as well as using strcpy() rather than assigning values to specific pointers directly.

shuffle_functions.c

This file contains functions for shuffling and output functions. There are 8 methods inside.

1. To obtain binary numbers from the deck size, I used the normal mathematical algorithm:
    a. Divide the decimal number by 2
    b. Get the remainder from A) and store it from the index 0 in an array. Meanwhile, the current index increments.
    c. The new decimal number should be the quotient. Repeat until the quotient becomes zero.

    The current index of the array is the length + 1 of the actual number of bits of the new binary number. Then we can trace back the array to obtain the inverse of the binary number, since the inverse should be the real binary number we need.

2. When shuffling cards, cards should be partitioned into two halves. The two empty array of pointers(referring to char* content) will carry only **contents** of top half cards and bottom half cards respectively after partition(). By imitating perfect shuffle in human's way, the current decklist will update its cards alternatively, either from the first_half or from the second_half, depending on the position and index. *The reason why I use arrays of pointers is* that firstly, I prefer to practice with 2-D arrays and proper double pointers. Secondly, since making two new decklists can occupy more memory and require more time to run, arrays of pointers to only store card contents can be efficient and clear.

```
char *first_half[deck_size / 2];
char *second_half[deck_size / 2];
int firstIndex = 0;
int secondIndex = 0;
card *current = cards->head; // as an
```

```
for (int i = 0; i < deck_size; i++)
{
    if (i % 2 == 0)
    { // extracts a card from the second_half decklis
        current->content = second_half[secondIndex];
        secondIndex++;
    }
    else
    {
        current->content = first_half[firstIndex];
        firstIndex++;
    }
    current = current->next; // the next card of the
}
```

3. *The problematic process is how to safely take cards to consist a new decklist.* I just change the content of each card to implement updating process to obtain a new decklist after each shuffle, but not making a new card with the new content value.

node_imple.c

This file contains all functions needed to implement a linked list, including adding a new node and creating a new list with basic initialisation.

```
 */
card *addNewNode(card *node, char *value) {
  card *c = malloc(sizeof(struct inode));
  if(!c){
    abort();
  }
  c->content = malloc(sizeof(strlen(value) + 1));
  if(!(c->content)){
    abort();
  } // add extra "1" to reserve space for '\0'
  strcpy(c->content, value);
  c->next = node; // usually NULL is passed by this variable
  return c;
}
```

1. *The problem here is given a char*, how can we use it as the content of the current card?* **Direct assignment does not work since it hits segmentation fault. We have to malloc() a certain and valid address to hold the value.**
2. *Direct assignment still does not work* - every card will hold the **same value**, since assignment just pass the **same address** to every card and the value in this address is changing for every new token. Eventually, every card will hold the value of the last token. Therefore, a **strcpy()** is useful to pass the value.

Testing

Stacscheck

```
acticals/Prac2-C2/tests
Testing CS2002 C2
- Looking for submission in a directory called 'Practical2': Already in it!
* BUILD TEST - build-clean : pass
* BUILD TEST - numerical/build-faro : pass
* COMPARISON TEST - numerical/prog-faro-num_36_22.out : pass
* BUILD TEST - ranksuit/build-faro : pass
* COMPARISON TEST - ranksuit/prog-faro-full_52_6.out : pass
* COMPARISON TEST - ranksuit/prog-faro-half_26_11.out : pass
6 out of 6 tests passed
```

Self-Testing

When a user inputs some invalid inputs, some proper warnings will be prompted.

```c
// check the validality of arguments
if (argc != 2)
{
  printf("Usage: ./faro_shuffle <arg> \n");
  exit(0);
}

if (!(strcmp(argv[1], "RANKSUIT") == 0 ||
      strcmp(argv[1], "NUMERICAL") == 0))
{
  printf("Usage: <arg> must be RANKSUIT or NUMERICAL \n");
  exit(0);
}
```

```
104
6
AC KC QC JC 0C 9C 8C 7C 6C 5C 4C 3C 2C AS KS QS JS 0S 9S 8S 7S 6S 5S 4S 3S 2S AD
 KD QD JD 0D 9D 8D 7D 6D 5D 4D 3D 2D AH KH QH JH 0H 9H 8H 7H 6H 5H 4H 3H 2H AC K
C QC JC 0C 9C 8C 7C 6C 5C 4C 3C 2C AS KS QS JS 0S 9S 8S 7S 6S 5S 4S 3S 2S AD KD
QD JD 0D 9D 8D 7D 6D 5D 4D 3D 2D AH KH QH JH 0H 9H 8H 7H 6H 5H 4H 3H 2H
IN : AC AC KC KC QC QC JC JC 0C 0C 9C 9C 8C 8C 7C 7C 6C 6C 5C 5C 4C 4C 3C 3C 2C
2C AS AS KS KS QS QS JS JS 0S 0S 9S 9S 8S 8S 7S 7S 6S 6S 5S 5S 4S 4S 3S 3S 2S 2S
 AD AD KD KD QD QD JD JD 0D 0D 9D 9D 8D 8D 7D 7D 6D 6D 5D 5D 4D 4D 3D 3D 2D 2D A
H AH KH KH QH QH JH JH 0H 0H 9H 9H 8H 8H 7H 7H 6H 6H 5H 5H 4H 4H 3H 3H 2H 2H EoD
IN : AD AC AD AC KD KC KD KC QD QC QD QC JD JC JD JC 0D 0C 0D 0C 9D 9C 9D 9C 8D
8C 8D 8C 7D 7C 7D 7C 6D 6C 6D 6C 5D 5C 5D 5C 4D 4C 4D 4C 3D 3C 3D 3C 2D 2C 2D 2C
 AH AS AH AS KH KS KH KS QH QS QH QS JH JS JH JS 0H 0S 0H 0S 9H 9S 9H 9S 8H 8S 8
H 8S 7H 7S 7H 7S 6H 6S 6H 6S 5H 5S 5H 5S 4H 4S 4H 4S 3H 3S 3H 3S 2H 2S 2H 2S EoD
OUT: AD AH AC AS AD AH AC AS KD KH KC KS KD KH KC KS QD QH QC QS QD QH QC QS JD
JH JC JS JD JH JC JS 0D 0H 0C 0S 0D 0H 0C 0S 9D 9H 9C 9S 9D 9H 9C 9S 8D 8H 8C 8S
 8D 8H 8C 8S 7D 7H 7C 7S 7D 7H 7C 7S 6D 6H 6C 6S 6D 6H 6C 6S 5D 5H 5C 5S 5D 5H 5
C 5S 4D 4H 4C 4S 4D 4H 4C 4S 3D 3H 3C 3S 3D 3H 3C 3S 2D 2H 2C 2S 2D 2H 2C 2S EoD
```

As we can see, the program can handle more than just 52 cards and still give a correct answer.

Extension

Numerical

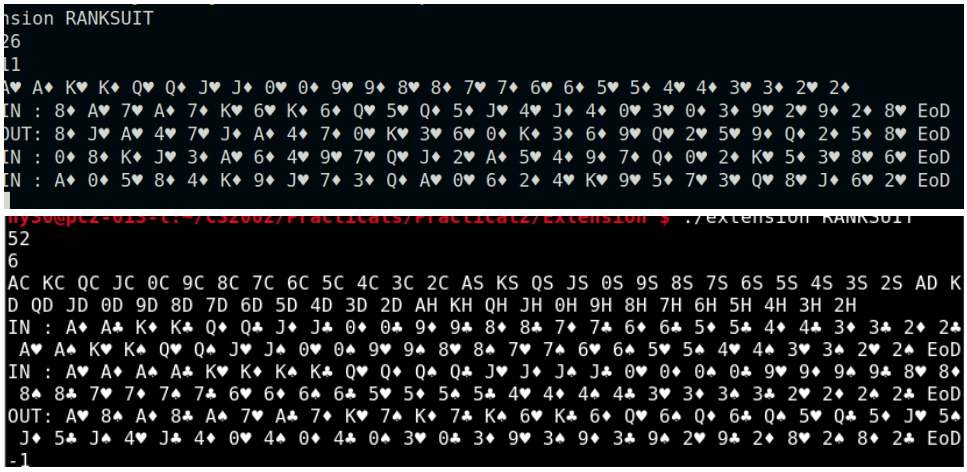
This extension requires students to provide functionality for decks of cards larger than 52 and the length of the value can be arbitrary. As the stacscheck shows, this extension is integrated into the main practical and works properly.

UTF-8 compliant program

This extension requires students to store chars and strings in UTF-8 and functions should be able to manipulate them to give a correct output. My implementation simply

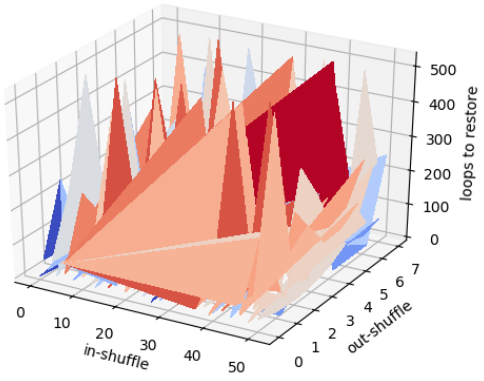
accepts normal characters inputs and UTF-8 inputs and convert them into the UTF-8 form. In this way, the output in the terminal still shows the correct contents after shuffle but represents each card value to meet the requirement.

In read_line() function, the value returned by getchar() will be checked. If the value describes a suit, then corresponding transformation will be invoked, and UTF-8 characters will be stored and displayed.



Fixed number to the original

We already know that some fixed number of out-shuffles or in-shuffles will lead a deck of cards to its original order. For out-shuffles, it is 8, and for in-shuffles, it is 52 (Diaconis, Graham and Kantor, 1983). So, I developed an experiment to find the fixed number of mix-shuffles to turn it back to the original. Additionally, **the obtained result will be written into a txt file. Those data will be extracted from the txt file into python for further analysis.** If the marker still has extra time, you can look at my python file to find the plotting process in ***plot.py***.



I have used FILE pointer to open *poker.txt* to extract all 52 card values and make a decklist consist of those 52 cards. Each time a mix-shuffle completes, the current decklist will be compared to the original decklist to see if they contain the same contents. If they do, then the value of the counter, the number of in-shuffles and the number of out-shuffles in a mix-shuffle will be record into "data.txt".

In axes, the reason why the range of in-shuffle is from 0 to 51 and the range of out-shuffle is from 0 to 7 is because 52 and 8 are fixed numbers that drive the decklist to the original. If we choose 52 in-shuffles and 2 out-shuffles, there is

no difference from 0 in-shuffles and 2 out-shuffles. Therefore, we only need to pick the range from 0 to 51 and the range from 0 to 7 to complete this experiment.

Testing: In data.txt, when a mix-shuffle contains 0 in-shuffle and 1 out-shuffle, it needs 8 mix-shuffles to recover; while a mix-shuffle contains 1 in-shuffle and 0 out-shuffle, it needs 52 to recover. This result generated by my algorithm has shown that my implementation is correct.

It shows that, for a certain mix-shuffle, the more in-shuffles and out-shuffles perform their functionality, the more mix-shuffles they need to go back to the original image. However, this is still a rough assumption and a tendency. The more complicated mathematical model can be solved by the group theory.

Evaluation & Conclusion

1. The full functionality of the standard specification satisfies requirements.
    a. All stacschecks have been passed.
    b. Codes are clear and simple to follow, with sufficient comments and explanation.
    c. There is no duplicate method and tedious method.
2. Extensions perform well according to further requirements.
    a. Extensions have passed the stacschecks.

In conclusion, during these two weeks, I have practiced C programming and acquire deeper understandings about how memory functions work. I also have deeper insights into all possible errors and faults coming from coding, and more importantly, how to handle them. However, I still need to self-learn the debugger and I believe it can be greatly beneficial in the forthcoming practical. Besides, the malloc() really relates to many strange errors, and I should focus on this to discover more.

Reference List

Diaconis, P., Graham, R. and Kantor, W. (1983). The mathematics of perfect shuffles. Advances in Applied Mathematics, 4(2), pp.175-196.