Practical 2 Report

Introduction

For this practical we had to implement the 'Faro Shuffle'. This involved reading in and storing a deck of cards from standard input. In order to achieve full marks for this practical, there were several constraints that had to be satisfied, primarily that were weren't to used any array-like structures or make use of contiguous dynamic memory.

How to Run

1. Navigate to Practical2 folder in a terminal.
2. Type `make all`
3. Type `./faro_shuffle` to run the program. faro_shuffle takes inputs in a format as described by the practical specification.

Design

In order to avoid array-based structures when storing the card input, the deck is stored as a linked list of 'Card' structures. With a cards position not being stored, but being implicit from its index in the linked list.

When taking in size and a position k, if k was larger than the size of the deck this could cause an error. To simply this, if k > size, then k is changed to k mod size to represent starting again from the top of a deck of cards.

In order to perform a faro shuffle on a linked list, that list has to be split at the midpoint. This is done by the split function. In addition there is no need for separate implementations of shuffling for IN or OUT shuffling, as by swapping which half of the deck is given as the first input, one can replicate IN *and* OUT shuffles with one function.

In order to allow for different implementations of the 'back-end', there is a separate `faro` function that calls the functions associated with shuffling the deck of cards according to the value k. This allows the main logic of faro_shuffle.c to remain the same if at a later date I wanted to change how the deck of cards is stored and shuffled

Extensions

For the numerical extension the card struct was given an extra field to contain the integer instead of rank and suit. In order to read this field, new functions for reading in the values and printing the deck out were made. However the shuffle and split functions could be reused at they were purely structural changes to the deck that did not account for values.

Implementation

Since it was not specified that k had to be smaller than the size of the deck, there is an added if statement that takes k mod size if k > size of deck. This could have instead been an assert statement that would throw an error but this method allows for more valid inputs.

decToBinary has to return both the binary representation of a value k and the length of that binary representation. To this end a variable – *k_length* – was passed in by reference and edited in the function, while the function itself returned the binary value of k.

When reading the cards into the deck, there is an initial fgetc because otherwise the following fgets reads from the wrong line and causes a segmentation fault.‘string’ is given `sizeof(wchar_t) * size) + (size * sizeof(char) * 2.` This value comes from one UTF-8 character to represent the suit (e.g. ♦) and 2 characters to represent to represent the rank and extra null character. Once the line is read in the string is split with strtok, the last token produced this way is again split to remove the newline character at the end of the string.

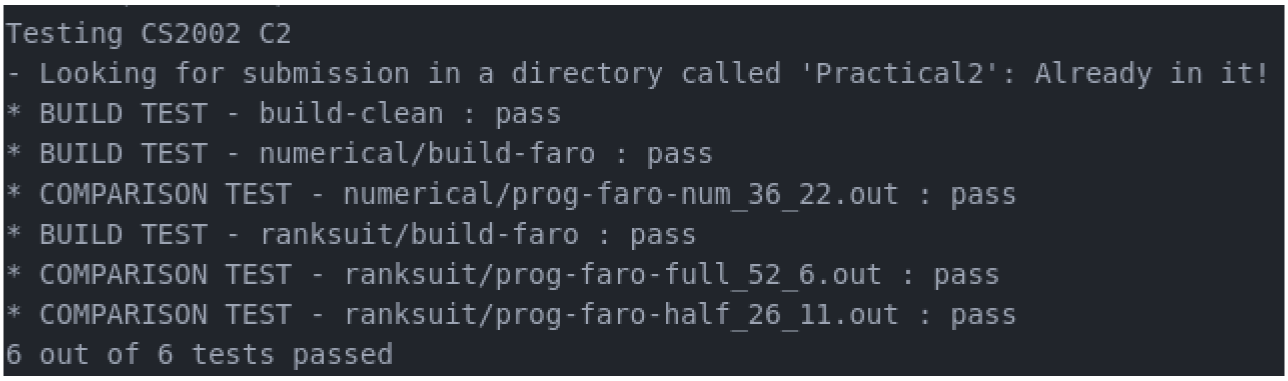
The faro function takes in the deck variable by reference. This is because the head node is moved around during the course of the faro function. If deck is not passed in by reference, then when it is freed at the end of the loop then the node head was pointing to could be at the end of linked list and caused a segmentation fault when being freed.

The shuffle function take one node from the beginning of the linked list ‘top’ then a node from the beginning of the linked list ‘bot’ repeating this process until a new linked list made from the interweaving of the two linked lists is made.

When freeing the memory from the linked list each node has to be iterated through individually and freed. There is additional complexity when freeing the values however. This is because each of values is actually just a pointer to position in an array of char* that contains the entire line of card values. Therefore to free the memory one needs to free the first value of the string which is now in position k. That means that position k has to be passed around a bit from main.c to free_ll.

Testing

StacsCheck



```
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
```

Basic Requirements

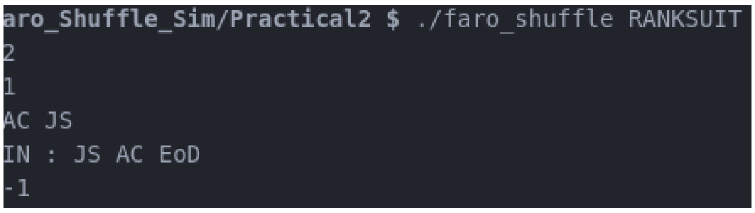
Test 1 – Small Data

Test Description: check that faro shuffle works with a deck of size 2.

Input: Size 2, Position: 1, AC JS

Expected Result: IN : JS AC EoD

Actual Result: Successful



```
aro_Shuffle_Sim/Practical2 $ ./faro_shuffle RANKSUIT
2
1
AC JS
IN : JS AC EoD
-1
```

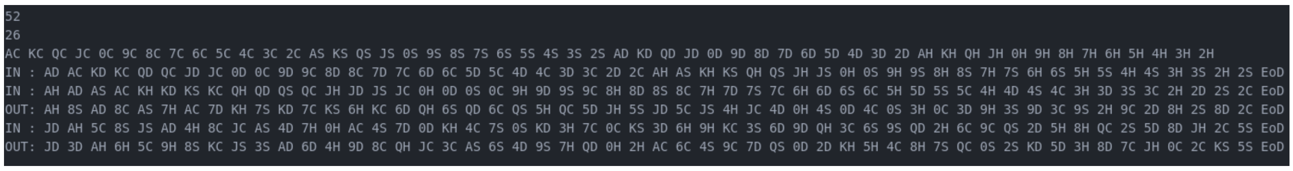
Test 2 – Large Data

Test Description: Check that faro shuffle works with a deck size of 52.

Input List: Size 52, Position 26 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

Expected List: AC in the 27th position

Actual Result: Success



```
52
26
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
IN : AD AC KD KC QD QC JD JC 0D 0C 9D 9C 8D 8C 7D 7C 6D 6C 5D 5C 4D 4C 3D 3C 2D 2C AH AS KH KS QH QS JH JS 0H 0S 9H 9S 8H 8S 7H 7S 6H 6S 5H 5S 4H 4S 3H 3S 2H 2S EoD
IN : AH AD AS AC KH KD KS KC QH QD QS QC JH JD JS JC 0H 0D 0S 0C 9H 9D 9S 9C 8H 8D 8S 8C 7H 7D 7S 7C 6H 6D 6S 6C 5H 5D 5S 5C 4H 4D 4S 4C 3H 3D 3S 3C 2H 2D 2S 2C EoD
OUT: AH 8S AD 8C AS 7H AC 7D KH 7S KD 7C KS 6H KC 6D QH 6S QD 6C QS 5H QC 5D JH 5S JD 5C JS 4H JC 4D 0H 4S 0D 4C 0S 3H 0C 3D 9H 3S 9D 3C 9S 2H 9C 2D 8H 2S 8D 2C EoD
IN : JD AH 5C 8S JS AD 4H 8C JC AS 4D 7H 0H AC 4S 7D 0D KH 4C 7S 0S KD 3H 7C 0C KS 3D 6H 9H KC 3S 6D 9D QH 3C 6S 9S QD 2H 6C 9C QS 2D 5H 8H QC 2S 5D 8D JH 2C 5S EoD
OUT: JD 3D AH 6H 5C 9H 8S KC JS 3S AD 6D 4H 9D 8C QH JC 3C AS 6S 4D 9S 7H QD 0H 2H AC 6C 4S 9C 7D QS 0D 2D KH 5H 4C 8H 7S QC 0S 2S KD 5D 3H 8D 7C JH 0C 2C KS 5S EoD
```

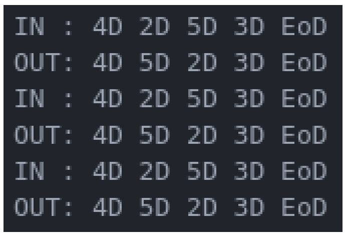
Test 3 – Repeated testing

Test Description: The program should keep looking for inputs until the size -1 is given.

Input: Size 4, position 2, list: 2D 3D 4D 5D, repeated 3 times.

Expected Result: The same set of outputs 3 times.

Actual Result: Success



```
IN : 4D 2D 5D 3D EoD
OUT: 4D 5D 2D 3D EoD
IN : 4D 2D 5D 3D EoD
OUT: 4D 5D 2D 3D EoD
IN : 4D 2D 5D 3D EoD
OUT: 4D 5D 2D 3D EoD
```

Extensions

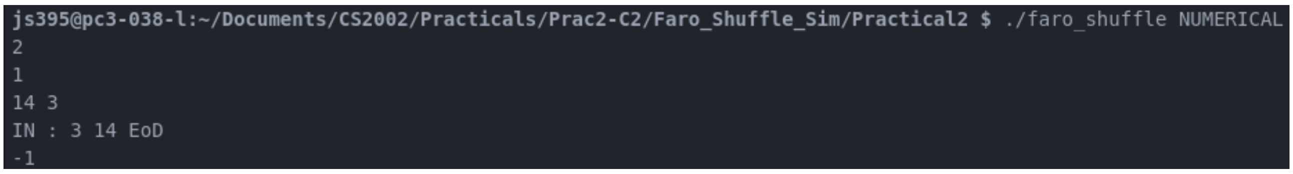
Test 4 – NUMERICAL small

Test Description: Check that faro shuffle works with the NUMERICAL option of size 2

Input List: Size 2, Position: 1, 14 3

Expected Result: IN: 3 14 EoD

Actual Result: Successful



```
js395@pc3-038-l:~/Documents/CS2002/Practicals/Prac2-C2/Faro_Shuffle_Sim/Practical2 $ ./faro_shuffle NUMERICAL
2
1
14 3
IN : 3 14 EoD
-1
```

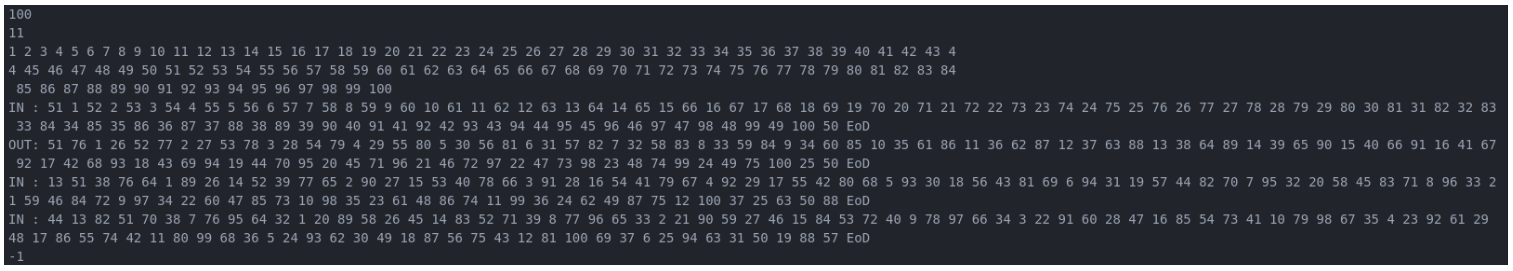
Test 5 – NUMERICAL Large

Test Description: Check that faro shuffle works with the NUMERICAL option of size 100

Input List: Size 100, Position: 11 – 1-100 in order.

Expected Result: 1 in the 12th position

Actual Result: Successful



```
100
11
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 4
4 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
IN : 51 1 52 2 53 3 54 4 55 5 56 6 57 7 58 8 59 9 60 10 61 11 62 12 63 13 64 14 65 15 66 16 67 17 68 18 69 19 70 20 71 21 72 22 73 23 74 24 75 25 76 26 77 27 78 28 79 29 80 30 81 31 82 32 83
 33 84 34 85 35 86 36 87 37 88 38 89 39 90 40 91 41 92 42 93 43 94 44 95 45 96 46 97 47 98 48 99 49 100 50 EoD
OUT: 51 76 1 26 52 77 2 27 53 78 3 28 54 79 4 29 55 80 5 30 56 81 6 31 57 82 7 32 58 83 8 33 59 84 9 34 60 85 10 35 61 86 11 36 62 87 12 37 63 88 13 38 64 89 14 39 65 90 15 40 66 91 16 41 67
 92 17 42 68 93 18 43 69 94 19 44 70 95 20 45 71 96 21 46 72 97 22 47 73 98 23 48 74 99 24 49 75 100 25 50 EoD
IN : 13 51 38 76 64 1 89 26 14 52 39 77 65 2 90 27 15 53 40 78 66 3 91 28 16 54 41 79 67 4 92 29 17 55 42 80 68 5 93 30 18 56 43 81 69 6 94 31 19 57 44 82 70 7 95 32 20 58 45 83 71 8 96 33 2
1 59 46 84 72 9 97 34 22 60 47 85 73 10 98 35 23 61 48 86 74 11 99 36 24 62 49 87 75 12 100 37 25 63 50 88 EoD
IN : 44 13 82 51 70 38 7 76 95 64 32 1 20 89 58 26 45 14 83 52 71 39 8 77 96 65 33 2 21 90 59 27 46 15 84 53 72 40 9 78 97 66 34 3 22 91 60 28 47 16 85 54 73 41 10 79 98 67 35 4 23 92 61 29
48 17 86 55 74 42 11 80 99 68 36 5 24 93 62 30 49 18 87 56 75 43 12 81 100 69 37 6 25 94 63 31 50 19 88 57 EoD
-1
```

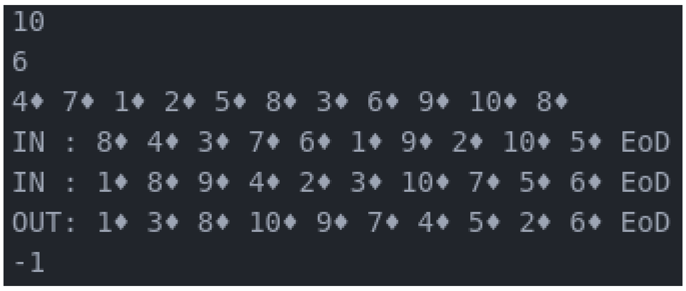
Test 6 – UTF-8

Test Description: Check that the RANKSUIT functionality works correctly with UTF characters

Input List: Size 10 position 6, 4♦ 7♦ 1♦ 2♦ 5♦ 8♦ 3♦ 6♦ 9♦ 10♦ 8♦

Expected Result: 4 should be in the 7th position.

Actual Result: Successful

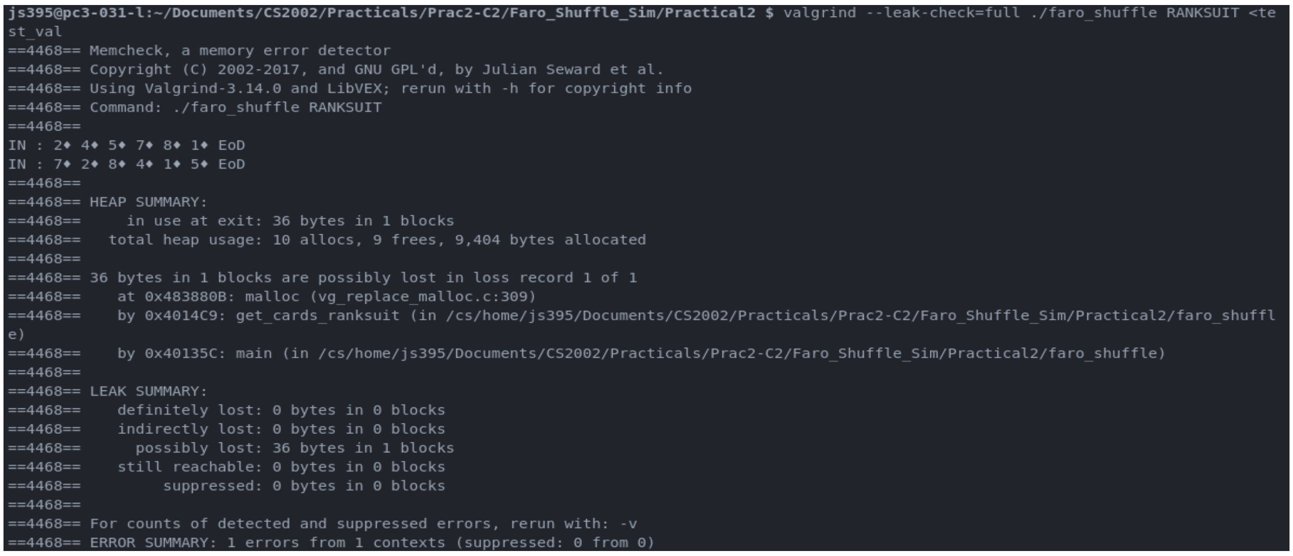


```
10
6
4♦ 7♦ 1♦ 2♦ 5♦ 8♦ 3♦ 6♦ 9♦ 10♦ 8♦
IN : 8♦ 4♦ 3♦ 7♦ 6♦ 1♦ 9♦ 2♦ 10♦ 5♦ EoD
IN : 1♦ 8♦ 9♦ 4♦ 2♦ 3♦ 10♦ 7♦ 5♦ 6♦ EoD
OUT: 1♦ 3♦ 8♦ 10♦ 9♦ 7♦ 4♦ 5♦ 2♦ 6♦ EoD
-1
```

Valgrind

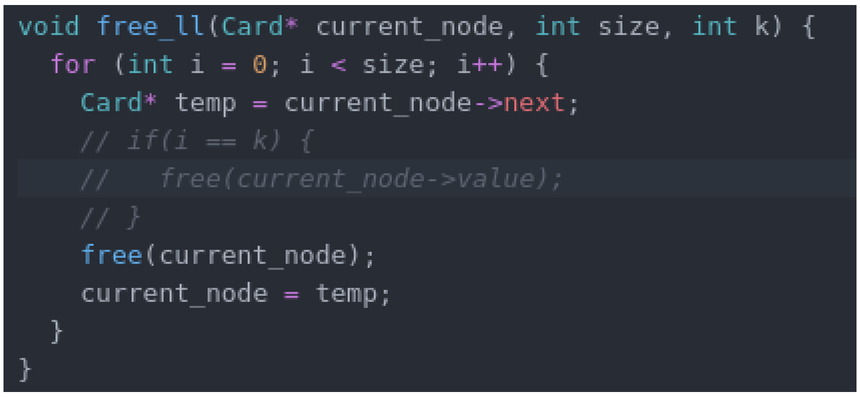
Valgrind it a debugging tool used to detect memory leaks in programs. It checks how much memory has been dynamically allocated and how much of it is freed before the program finishes execution. By entering the command valgrind <program> <commandline arguments> it can be run.

Below we can see an example run where some memory hasn't been freed, and the –leak-check=full argument is passed in.



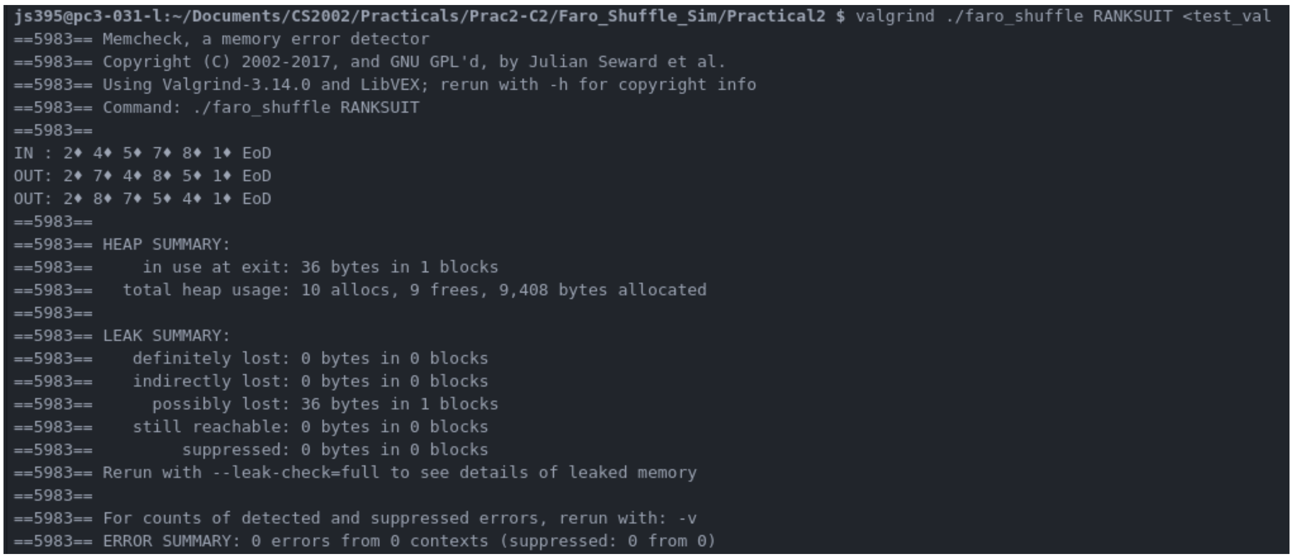
```
js395@pc3-031-l:~/Documents/CS2002/Practicals/Prac2-C2/Faro_Shuffle_Sim/Practical2 $ valgrind --leak-check=full ./faro_shuffle RANKSUIT <te
st_val
==4468== Memcheck, a memory error detector
==4468== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4468== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==4468== Command: ./faro_shuffle RANKSUIT
==4468==
IN : 2♦ 4♦ 5♦ 7♦ 8♦ 1♦ EoD
IN : 7♦ 2♦ 8♦ 4♦ 1♦ 5♦ EoD
==4468==
==4468== HEAP SUMMARY:
==4468==     in use at exit: 36 bytes in 1 blocks
==4468==   total heap usage: 10 allocs, 9 frees, 9,404 bytes allocated
==4468==
==4468== 36 bytes in 1 blocks are possibly lost in loss record 1 of 1
==4468==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==4468==    by 0x4014C9: get_cards_ranksuit (in /cs/home/js395/Documents/CS2002/Practicals/Prac2-C2/Faro_Shuffle_Sim/Practical2/faro_shuffl
e)
==4468==    by 0x40135C: main (in /cs/home/js395/Documents/CS2002/Practicals/Prac2-C2/Faro_Shuffle_Sim/Practical2/faro_shuffle)
==4468==
==4468== LEAK SUMMARY:
==4468==    definitely lost: 0 bytes in 0 blocks
==4468==    indirectly lost: 0 bytes in 0 blocks
==4468==      possibly lost: 36 bytes in 1 blocks
==4468==    still reachable: 0 bytes in 0 blocks
==4468==         suppressed: 0 bytes in 0 blocks
==4468==
==4468== For counts of detected and suppressed errors, rerun with: -v
==4468== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

This tells us exactly which function isn't freeing all its memory. In this example it is the get_cards_ranksuit function, which isn't freeing the memory allocated to the 'string' variable. If the free in free_ll is uncommented then all the memory should be freed successfully as seen below.



```c
void free_ll(Card* current_node, int size, int k) {
  for (int i = 0; i < size; i++) {
    Card* temp = current_node->next;
    // if(i == k) {
    //   free(current_node->value);
    // }
    free(current_node);
    current_node = temp;
  }
}
```

Valgrind is also useful for identifying when memory is being freed but then read at a



```
js395@pc3-031-l:~/Documents/CS2002/Practicals/Prac2-C2/Faro_Shuffle_Sim/Practical2 $ valgrind ./faro_shuffle RANKSUIT <test_val
==5983== Memcheck, a memory error detector
==5983== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5983== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==5983== Command: ./faro_shuffle RANKSUIT
==5983==
IN : 2♦ 4♦ 5♦ 7♦ 8♦ 1♦ EoD
OUT: 2♦ 7♦ 4♦ 8♦ 5♦ 1♦ EoD
OUT: 2♦ 8♦ 7♦ 5♦ 4♦ 1♦ EoD
==5983==
==5983== HEAP SUMMARY:
==5983==     in use at exit: 36 bytes in 1 blocks
==5983==   total heap usage: 10 allocs, 9 frees, 9,408 bytes allocated
==5983==
==5983== LEAK SUMMARY:
==5983==    definitely lost: 0 bytes in 0 blocks
==5983==    indirectly lost: 0 bytes in 0 blocks
==5983==      possibly lost: 36 bytes in 1 blocks
==5983==    still reachable: 0 bytes in 0 blocks
==5983==         suppressed: 0 bytes in 0 blocks
==5983== Rerun with --leak-check=full to see details of leaked memory
==5983==
==5983== For counts of detected and suppressed errors, rerun with: -v
==5983== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

later time. This came useful during this practical as in early implementations the memory containing the card values was freed at the end of the get_card_ranksuit function, but the pointers to those values were still being read. While this gave the correct values for later functions as the memory hadn't been overwritten at the time, this was not guaranteed. Valgrind identified this incorrect read operation.

Evaluation

This practical tested my knowledge of pointers, malloc and structs quite a lot. The part I found most difficult was making the program UTF-8 compliant as it required understanding how UTF characters are stored and allocating sufficient memory so that a char* could hold them.

When reading in values I had to precede reading the actual values with a fgetc, I am not sure why this is but the fgets wouldn't read from the correctly line without it.

Freeing memory that was split using the strtok function proved somewhat of a challenge as it wasn't immediately obvious how to free it as the first memory location was moved around due to the shuffling of the linked list.

This practical also put to use our knowledge of makefiles as as without them testing and making multiple implementations of the same functionality would have been difficult.

Conclusion

This practical was an interesting foray into dynamic memory allocation in C and tested my knowledge of various aspects of C such as structs, malloc, valgrind and composition of files in c.