CS2100 - Tutorial 2 - C: Pointers, Functions, Arrays and Structures

Week 4

1. Arrays

(a)

If limit > size of array, ⇒ seg fault

```c
int readArray(int arr[], int limit)
//Purpose: Read at most "limit" number of positive integers into arr
//         User can terminate by entering a negative integer
{
    int numOfItems = 0;
    int input;

    while (numOfItems < limit)
    {
        printf("Input: ");
        scanf("%d", &input);
        if (input < 0)
        {
            break;
        }
        arr[numOfItems++] = input;
    }

    return numOfItems;
}
```

(b).

```c
// Iterative version
// Other solutions possible
void reverseArray(int arr[], int size)
//Purpose: Reverse the items in array arr
{
    int temp;
    int left = 0;
    int right = size - 1;

    while (left < right)
    {
        temp = arr[left];
        arr[left++] = arr[right];
        arr[right--] = temp;
    }
}
```

(c)(i). Similar to iterative soln (swap extreme ends, move further in the middle until left ≥ right)

(ii). Need to pass in the arguments left = 0, right = size - 1

i) How does reverseArrayRec() work?

Using two indices left and right to "Shrink down the array" at every recursion

Once left > right i.e. condition not met, recursion stops

Ii) Why is the reverseArrayWrapper() needed?

Original reverseArray() function in Q1b only takes in array and size as parameters

Wrapper function essentially hides the change in parameters (abstraction) and presents the same interface to user

Kicks off the actual recursion by giving the initial left and right indices

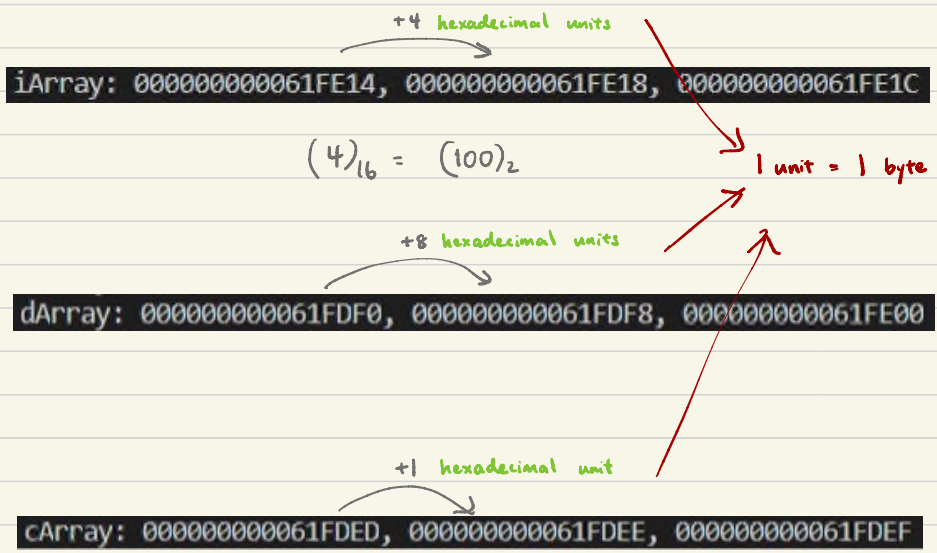
(d).

```
//Recursive solution WITHOUT wrapper
void reverseArrayRec2( int arr[], int size)
{
        int temp;

     if (size <= 1)
             return;

        temp = arr[0];
        arr[0] = arr[size-1];
        arr[size-1] = temp;
        reverseArrayRec2(&arr[1], size-2);
}
```

arrays in C are essentially pointers

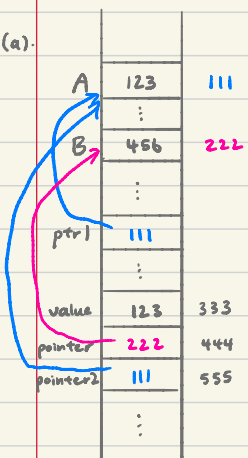
2. Arrays and memory

```c
#include <stdio.h>

int main()
{
    int iArray[3];
    double dArray[3];
    char cArray[3];

    printf("iArray: %p, %p, %p\n", &iArray[0], &iArray[1], &iArray[2]);
    printf("dArray: %p, %p, %p\n", &dArray[0], &dArray[1], &dArray[2]);
    printf("cArray: %p, %p, %p\n", &cArray[0], &cArray[1], &cArray[2]);
}
```



3. Parameter Passing

(a).



(b)(i). No. Pass-by value ⇒ different memory address

(ii). Yes. *pointer = <newValue>

Dereference

(iii). Yes. *pointer2 = <newValue>

(iv). No. Different memory address.

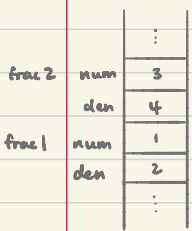
To change : use parameter **int ptrptr, then pass in &ptr1 as argument

Answers to 4(b), (d) in the back

4. Structures

(a).

```
In main(), frac1 is at 0x61fe18, frac2 is at 0x61fe10
The address of frac1.num is 0x61fe18, frac1.den is 0x61fe1c
```



(b).

```c
8   void swapFractionByValue( struct Fraction a, struct Fraction b)
9   //Purpose: To swap the content of a and b
10  {
11      int tempNum = a.num;
12      int tempDen = a.den;
13
14      a.num = b.num;
15      a.den = b.den;
16
17      b.num = tempNum;
18      b.den = tempDen;
19
20      printf("By Value parameter addressses:\n");
21      printf("a.num: 0x%x, a.den: 0x%x\n", &a.num, &a.den);
22      printf("b.num: 0x%x, b.den: 0x%x\n", &b.num, &b.den);
23  }
```

(c).

```
At the start: fract 1 is (1 / 2), frac2 is (3 / 4)
By Value parameter addressses:
a.num: 0x61fdf0, a.den: 0x61fdf4
b.num: 0x61fdf8, b.den: 0x61fdfc
After swapFractionByValue(): fract 1 is (1 / 2), frac2 is (3 / 4)
```

(d).

```
25  void swapFractionByAddress( struct Fraction* a, struct Fraction* b)
26  //Purpose: To swap the content of a and b
27  {
28      int tempNum = a->num;
29      int tempDen = a->den;
30
31      a->num = b->num;
32      a->den = b->den;
33
34      b->num = tempNum;
35      b->den = tempDen;
36
37      printf("By Address parameter addressses:\n");
38      printf("a.num: 0x%x, a.den: 0x%x\n", &(a->num), &(a->den));
39      printf("b.num: 0x%x, b.den: 0x%x\n", &(b->num), &(b->den));
40  }
```

(e).

```
By Address parameter addressses:
a.num: 0x61fe18, a.den: 0x61fe1c
b.num: 0x61fe10, b.den: 0x61fe14
After swapFractionByAddress(): fract 1 is (3 / 4), frac2 is (1 / 2)
```

matching memory addresses

```
In main(), frac1 is at 0x61fe18, frac2 is at 0x61fe10
The address of frac1.num is 0x61fe18, frac1.den is 0x61fe1c
```

Q4(b).

```
void swapFractionByValue( struct Fraction a, struct
Fraction b)
//Purpose: To swap the content of a and b
{
    struct Fraction temp;

    //printf() for part (c)
    //One way to show why this function wont work
properly is to show that the a and b
    // are distinct from the original fractions
    printf("In swapFractionByValue(), a is at 0x%x, b is
at 0x%x\n", &a, &b);

    temp = a;
    a = b;
    b = temp;
}
```

(d).

```
void swapFractionByAddress( struct Fraction* a, struct
Fraction* b)
//Purpose: To swap the content of a and b
{
    struct Fraction temp;

    //printf() for part (e)
    //take note of the differences between "a" (where a
points to) and "&a" (where is a itself)
    printf("In swapFractionByAddress(), a points to 0x%x,
b points to 0x%x\n", a, b);


    temp = *a;   //structure can be copied via assignment
    *a = *b;
    *b = temp;
}

//Additional functions carried from lecture 06. Provided
for you to play around if needed.

int GCD( int x, int y)
{
    if (x % y == 0){
        return y;
    }

    return GCD( y, x % y );
}
```