Lab 3: Memory Organization

1st implementation:

two_d_alloc allocates memory for a two dimensional array using the parameters for number of rows and columns (int n and int m respectively), and the size (int size) of each element in bytes.

two_d_dealloc frees up the memory of the given character-pointer.

two_d_store stores an integer (int x) into the array, 1 byte at the time. It does this by looking at 8 bits (1 byte) of the integer at a time, storing each byte into an element of the array. To make it possible to fetch the inserted value, we store each byte in succession on the same row (in row-major format; in column-major format we simply swap the rows with columns).

two_d_fetch looks at each element in the array and copies their values into a new array, each in their own element, then unionizes the 4 elements into a single integer and returns it.

void main holds the values for number of rows and columns, the size each element will be. The first double for-loop stores the values to the array. *i* represents the row and *j* represents the column. *j* is incremented by 4 each loop since each integer stored will take up 4 elements (because we are using row-major format). In a similar fashion, the second double for-loop fetches and prints out the values.

Row-major format

```
  GNU nano 2.2.6                File: task1.c

#include <stdio.h>
#include <stdlib.h>

char* two_d_alloc(int n, int m, int size) {
        char* myArray = (char*)malloc(n*m*sizeof(size));
        printf("allocated\n");
        return myArray;
}

void two_d_dealloc(char* myArray) {
        free(myArray);
        printf("deallocated\n");
}

void two_d_store(char* myArray, int size, int n, int m, int i, int j, int x) {
        *(myArray + i*m + j+3) = (x >> 6*size) & 0xFF;
        *(myArray + i*m + j+2) = (x >> 4*size) & 0xFF;
        *(myArray + i*m + j+1) = (x >> 2*size) & 0xFF;
        *(myArray + i*m + j) = x & 0xFF;
}
```

```
int two_d_fetch(char* myArray, int int_size, int n, int m, int i, int j) {
        char a[4];
        a[0] = *(myArray + i*m + j);
        a[1] = *(myArray + i*m + j+1);
        a[2] = *(myArray + i*m + j+2);
        a[3] = *(myArray + i*m + j+3);
        int value = *(int *)a;
        return value;
}
```

```
void main() {
        int n = 4;
        int m = 8;
        int size = sizeof(int);
        //printf("size: %d\n", size);
        char* d = two_d_alloc(n, m, size);

        int i,j,count = 0;
        for (i = 0; i < n; i++) {
                for (j = 0; j < m; j=j+4) {
                        two_d_store(d, size, n, m, i, j, count);
                        count = count + 1;
                }
        }
```

```
        int x;
        for (i = 0; i < n; i++)
                for (j = 0; j < m; j=j+4) {
                        x = two_d_fetch(d, size, n, m, i, j);
                        printf("The value in the position [%d %d] is: %d\n",
                                i+1, j+1, x);
                }
        two_d_dealloc(d);
}
```

Column-major format

```
void two_d_store(char* myArray, int size, int n, int m, int i, int j, int x) {
        *(myArray + j*n + i+3) = (x >> 6*size) & 0xFF;
        *(myArray + j*n + i+2) = (x >> 4*size) & 0xFF;
        *(myArray + j*n + i+1) = (x >> 2*size) & 0xFF;
        *(myArray + j*n + i) = x & 0xFF;
}

int two_d_fetch(char* myArray, int int_size, int n, int m, int i, int j) {
        char a[4];
        a[0] = *(myArray + j*n + i);
        a[1] = *(myArray + j*n + i+1);
        a[2] = *(myArray + j*n + i+2);
        a[3] = *(myArray + j*n + i+3);
        int value = *(int *)a;
        return value;
}
```

```
void main() {
        int n = 4;
        int m = 8;
        int size = sizeof(int);
        //printf("size: %d\n", size);
        char* d = two_d_alloc(n, m, size);
        int *value;

        int i,j,count = 0;
        value = &count;
        for (i = 0; i < n; i++) {
                for (j = 0; j < m; j=j+4) {
                        two_d_store(d, size, n, m, i, j, value);
                        count = count + 1;
                }
```

```
        int i,j,count = 0;
        for (j = 0; j < n; j++) {
                for (i = 0; i < m; i=i+4) {
                        two_d_store(d, size, n, m, i, j, count);
                        count = count + 1;
                }
        }
        int x;
        for (j = 0; j < n; j++)
                for (i = 0; i < m; i=i+4) {
                        x = two_d_fetch(d, size, n, m, i, j);
                        printf("The value in the position [%d %d] is: %d\n",
                                i+1, j+1, x);
                }
```

```
        //int *x;
        for (i = 0; i < n; i++)
                for (j = 0; j < m; j=j+4) {
                        //x = two_d_fetch(d, size, n, m, i, j);
                        printf("The value in the position [%d %d] is: %d\n",
                                i+1, j+1, *(two_d_fetch(d, size, n, m, i, j)));
                }
        two_d_dealloc(d);
}
```

Output

```
pi@raspberrypi:~ $ gcc -o task1_2 task1_2.c
pi@raspberrypi:~ $ ./task1_2
allocated
The value in the position [1 1] is: 0
The value in the position [1 2] is: 1
The value in the position [1 3] is: 2
The value in the position [1 4] is: 3
The value in the position [1 5] is: 4
The value in the position [1 6] is: 5
The value in the position [1 7] is: 6
The value in the position [1 8] is: 7
deallocated
pi@raspberrypi:~ $ _
```

2nd implementation

two_d_alloc allocates memory for a two dimensional array using the number of rows and columns given. The size of the elements in the array is given by the size of the arg, which is defined with a type definition - *typedef double arg* - so that we can change the type of the elements that we store in the array.

two_d_store stores a double (arg x) into the array. However, unlike the previous implementation pointers and arg-pointers are used to access the addresses of the elements and store them in the array.

two_d_fetch creates an arg-pointer and stores the desired element in that pointer and returns it.

void main creates a char-pointer two-dimensional array which is sent into two_d_store. An arg variable called store is also created and set equal to 1.1. Through the iterations store is incremented by 1 each time. Store is sent into two_d_store as the element to be placed in the array. Another iteration fetches the elements in the array and prints them out sequentially.

```c
#include <stdio.h>
#include <stdlib.h>

typedef double arg;

union myUnion {
    int i;
    float f;
    double d;
} ;

char* two_d_alloc(int n, int m, int size) {
    char* myArray = (char*)malloc(n*m*sizeof(size));
    printf("Allocated\n");
    return myArray;
}

void two_d_dealloc(char* myArray) {
    free(myArray);
    printf("Deallocated\n");
}
```

```c
void two_d_store(char* myArray, int size, int n, int m, int i, int j, arg *x) {

    *(arg*)(myArray + (i*m + j)*size) = *x;
}

arg* two_d_fetch(char* myArray, int size, int n, int m, int i, int j) {

    arg * p;
    p = (arg*)(myArray + (i*m + j)*size);

    return p;
}
```

```c
int main() {
    int n = 4;
    int m = 8;
    int size = sizeof(arg);
    char* myArray = two_d_alloc(n,m,size);

    arg store = 1.1;
    int i,j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j = j + 4) {
            two_d_store(myArray, size, n, m, i, j, &store);
            store = store + 1;
        }
    }

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j = j + 4) {
            printf("The value in position [%d %d] is: %g\n", i+1, j+1, *(two_d_fetch(myArray, size, n, m, i, j)));
        }
    }
    two_d_dealloc(myArray);

    return 0;
}
```

Output

```
The value in position [1 1] is: 1.1
The value in position [1 5] is: 2.1
The value in position [2 1] is: 3.1
The value in position [2 5] is: 4.1
The value in position [3 1] is: 5.1
The value in position [3 5] is: 6.1
The value in position [4 1] is: 7.1
The value in position [4 5] is: 8.1
Deallocated
```