Arash Mostafaee
Foad Ghafourian

Lab 3: Memory Organization

The main focus of this lab is how to use a 1D character buffer in order to emulate 2D arrays in C.

Task 1

The task is to implement storage and fetching in a 2D byte array by following several steps.

The first step is to implement a function “two_d_alloc that” allocates char* to store an array, the second step is to deallocate the array. “r” is the integer size of row and “c” is the column. In figure 1.1, we allocate a block of memory and deallocate the memory when its no longer needed.

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

char *two_d_alloc (char *a, int r, int c, int size){
        a = malloc(r*c*size);
        //printf("address after allocation %p \n", a);
}

char *two_d_dealloc(char *a){
        free(a);
        //printf("after deallocation %p \n", a);
}
```

Figure 1.1

After our initial step we needed to implement a function “two_d_store” which stores an integer argument into a particular row and column of the array. Here it was important use bit shift, especially right shift which the value is moved right by the number of bits specified by the right operand. In addition, assuming its row major order, is done by the row(where the element is) multiplied by the number of columns in the array added to the column(where the element is), r+c1+c.

We also need a function which return the value of the integer argument stored in the memory array. Since we did a right bitwise shift when we stored, we need to bit shift to the left in order to get the correct value when we fetch the data.

```
void two_d_store(char *a,int size, int r, int c, int r1, int c1, int value){
        int dest = (r*c1+c)*size; int i = 0;
        for(i; i < size; i++){
                a[dest+i] = value >> 8*i;
                //store integer argument and bitwise right shift
                printf("%x %x %x %x \n", a[dest+3], a[dest+2], a[dest+1],
                a[dest]);
        }
}

int two_d_fetch(char *a,int size, int r, int c, int r1, int c1){
        int dest = (r*c1 + c)*size;
        int fetch =
        (a[dest+3]<<24)|
        (a[dest+2]<<16)|
        (a[dest+1]<<8)|
        (a[dest]);
        //to find a start byte to fetch data
        //bit shift data to the left, reverse from the previous function
        return fetch;
}
```

Figure 1.2

To rewrite our functions to use column major order, we only need to change the formula of row major order (figure 1.3). The column(where the element lies) multiplied by the number of rows in the array added by the row (where the element lies).

```
void store_column(char *a, int size, int value, int r, int c, int r1, int c1){
        int dest = (r1*c+r)*size;
        int i = 0;
        for(i; i<size; i++){
                a[dest+i] = value >> 8*i;
        }
}

int fetch_column(char *a, int size, int r, int c, int r1, int c1){
        int dest = (r1*c+r)*size;
        int value = (a[dest+3]<24) | (a[dest+2]<<16) |
        (a[dest+1]<<8) | (a[dest]);
        return value;
}
```

Figure 1.3

In figure 1.4, we have implemented a verison of “two_d_store” and “two_d_fetch” which uses the memory address of arguments. The comments in the figure explains how this is done.

```
void store_memory(char *a, int size, char *value, int r, int c, int r1, int c1){
        int dest = (r*c1+c)*size;
        int i = 0;
        for(i; i < size; i++){
                a[dest+i] = value[i];
                //access first byte in position and save data in 'a'
        }
}

char *fetch_memory(char *a, int size, int r, int c, int r1, int c1){
        int dest = (r*c1+c)*size;
        char *value = &(a[dest]);
        //return address of first byte in position
        //and return address to original type
}
```

Figure 1.4

The last step of the task was to use the memory address to an arbitrary data type. Figure 1.5 shows how this is done.

```
int main(){
        char *a;
        int r1 = 2; int c1 = 19; int size = 8;
        a = two_d_alloc(a,r1,c1,size);
        double value = 123;
        store_memory(a,size,(char*)&value ,0,0,r1,c1);
        double *ans = (double*)(fetch_memory(a,size,0,0,r1,c1));
        printf("fetch memory double: %f\n", *ans);
        two_d_dealloc(a);
//      memory_dump(a, size, r1, c1);
}
```

```
[ Wrote 91 lines ]

pi@raspberrypi:~ $ gcc lab3t1.c -o orre
pi@raspberrypi:~ $ ./orre
fetch memory double: 123.000000
pi@raspberrypi:~ $
```

Figure 1.5

Arash Mostafaee
Foad Ghafourian

Task 2

With this task, our thought was to go through each character in each string since it wasn't possible to convert a string into hexadecimal notation. As it can be seen in the first loop, we try to go through the bytes in the array and in the second loop, we try to go through each of the strings.

```
void memory_dump(char *a, int size, int r1, int c1){
        int i = 0;
        for (i; i < r1*c1*size; i=i+size){
                if(i % (4*size) == 0)
                printf("\n %p - ", &a[i]);
                int j = 0;
                for(j; j < size; j++){
                        printf("%x", a[i+j]);
                }
                printf(" - ");
        }
}
```

The issue with our output is that we forgot to implement the string in our main function.

```
pi@raspberrypi:~ $ gcc lab3t1.c -o orre
pi@raspberrypi:~ $ ./orre

 0x21008 - 00000c05e40 - 00000000 - 00000000 - 00000000 -
 0x21028 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x21048 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x21068 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x21088 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x210a8 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x210c8 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x210e8 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x21108 - 00000000 - 00000000 - 00000000 - 00000000 -
 0x21128 - 00000000 - 00000000 - pi@raspberrypi:~ $
```