

Lab 3

Humam Amouri & Yonis Said



The following laboration was done in QEMU virtual machine using C programming.

The first task at hand was called "Array storage" and its purpose was to use a 1D character buffer to emulate a 2D array. The requirements were array storage, memory addressing and fetching addresses/values. The details included also implementations using column/row-major ordering.

We started by allocating and deallocating memory by creating the following two functions (see *code snippet 1*):

- `two_d_alloc`: using "malloc" we allocated memory for our 2D array. This was done by calculating the number of elements in the 2D array and multiplying it by the desired size. To access the allocated memory we return our character array.
- `two_d_dealloc`: deallocating memory was accomplished by using the function "free".

After allocating memory we need to be able to store and fetch data from our 1D array. Using row-major order we can store and fetch by using the following two function:

- `Two_d_store`: stores an integer argument into a particular row and column of the array. As shown in the *code snippet* below our value can be stored by finding the position which projects the 2D coordinates to 1D coordinates. After finding the correct starting byte we start bit shifting to the right so that each byte is stored in its respective place (check the blue commented text in *code snippet 1* for more details). To check that data was stored correctly we print out all the allocated bytes.

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

char* two_d_alloc(char *myarr, int row, int column, int size) {
    myarr = malloc(row*column*size);

    return myarr;
}

void two_d_dealloc(char* myarr){
    free(myarr);
}

void two_d_store(char *myarr,int size, int value, int row, int column, int numr,
int numc){
    int position = (row*numc+column)*size;
    int i=0;
    for (i;i<size;i++) myarr[position+i] = value >>8*i;
/*
    myarr[position+1] = value >>8;
    myarr[position+2] = value >>16;
    myarr[position+3] = value >> 24;
*/
    printf("%x %x %x %x \n",myarr[position+3],myarr[position+2],
    myarr[position+1],myarr[position]);
}
```

Code snippet 1

- `two_d_fetch`: to get back our stored data we need to think backwards. So this function starts by finding a starting byte to fetch the data from. Then it bit-shifts the data to the left in a reverse manner to the function `two_d_store`. The bit-shifted data is then summed to give us our final value which we return.

```
int two_d_fetch(char *myarr,int size,int row, int column, int numr,int numc ){
    int position = (row*numc+column)*size;
    int value = (myarr[position+3]<<24) | (myarr[position+2]<<16) |
    (myarr[position+1]<<8) | (myarr[position]);

    return value;
}
```

Code snippet 2

We've verified that the implemented functions work as expected by allocating a 2D array, storing and then fetching a number of integer arguments. We've even tested the most common special cases and they seem to work (see *code snippet 3*).

```
char *myarr;
int numr =2; int numc =19;
int size = 4;
myarr = two_d_alloc(myarr,numr,numc,size);
two_d_store(myarr,size,5,0,0,numr,numc);
two_d_store(myarr,size,300,1,2,numr,numc);
two_d_store(myarr,size,255,0,8,numr,numc);
two_d_store(myarr,size,-1,1,10,numr,numc);
printf("two d fetch: %d\n",two_d_fetch(myarr,size,0,0,numr,numc));
printf("two d fetch: %d\n",two_d_fetch(myarr,size,1,2,numr,numc));
printf("two d fetch: %d\n",two_d_fetch(myarr,size,0,8,numr,numc));
printf("two d fetch: %d\n",two_d_fetch(myarr,size,1,10,numr,numc));
two_d_dealloc(myarr);
/*
char *value = "a";
int row = 1;
int column = 2;
[ Wrote 160 lines ]
```

```
pi@raspberrypi:~ $ gcc array.c -o ra
./ra
pi@raspberrypi:~ $ ./ra
5 0 0 0
2c 1 0 0
ff 0 0 0
ff ff ff ff
two d fetch: 5
two d fetch: 300
two d fetch: 255
two d fetch: -1
pi@raspberrypi:~ $
```

Code snippet 3

The next required step from us was to rewrite our `two_d_store` and `two_d_fetch` functions so that they use a column-major format instead of the row-major one they were using. What we needed to change in both functions was to change the position so that it uses column ordering (see *code snippet 4*).

```
void two_d_store_column(char *myarr,int size, int value, int row, int column,
int numr, int numc){
    int position = (column*numr+row)*size;
    int i=0;
    for (i;i<size;i++) myarr[position+i] = value >>8*i;
}

int two_d_fetch_column(char *myarr,int size,int row, int column, int numr,
int numc ){
    int position = (column*numr+row)*size;
    int value = (myarr[position+3]<<24) | (myarr[position+2]<<16) |
(myarr[position+1]<<8) | (myarr[position]);
    return value;
}
```

Code snippet 4

The last part of task 1 wanted us to store and fetch any type of data type in our char array by using its array-address instead of value. The key concept of this problem is to work with pointers instead of direct values and because our array has the type “char” then we must cast over all input value addresses which doesn’t have the type “char”. This means that the process of storing the value will be expressed like this:

- Create value
- Send in the value’s address after casting it to a “char” pointer
- Use the address to access the first byte in our position and save the data in the array

And fetching:

- Return the address of the first byte in our position
- Cast the char pointer pointing at the returned address to its original data type.

The following code snippet shows how both functions were written:

```
void two_d_store_mem(char *myarr,int size, char *value, int row, int column,
int numr, int numc){
    int position = (row*numc+column)*size;
    int i =0;
    for (i;i<size;i++) myarr[position+i] = value[i];
}

char* two_d_fetch_mem(char *myarr,int size,int row, int column, int numr,
int numc ){
    int position = (row*numc+column)*size;
    char* value = &(myarr[position]);

    return value;
}
```

Code snippet 5

While the following code snippet shows how the input looks like and what the output is for a double input:

```
int main(){
    char *myarr;
    int numr =2; int numc =19;
    int size = 8;
    myarr = two_d_alloc(myarr,numr,numc,size);
    double value =55555555555;
    two_d_store_mem(myarr,size,(char*)&value ,0,0,numr,numc);
    double* ans =(double*)(two_d_fetch_mem(myarr,size,0,0,numr,numc));
    printf("two d fetch mem double: %f\n",*ans);
    two_d_dealloc(myarr);
    /*
    char *value = "a";
    int i=0;
    for (i; i < numr;i++ ){
        [ Wrote 156 lines ]
    }
    */
}

pi@raspberrypi:~ $ gcc array.c -o ra
./ra
pi@raspberrypi:~ $ ./ra
two d fetch mem double: 5555555555.000000
pi@raspberrypi:~ $
```

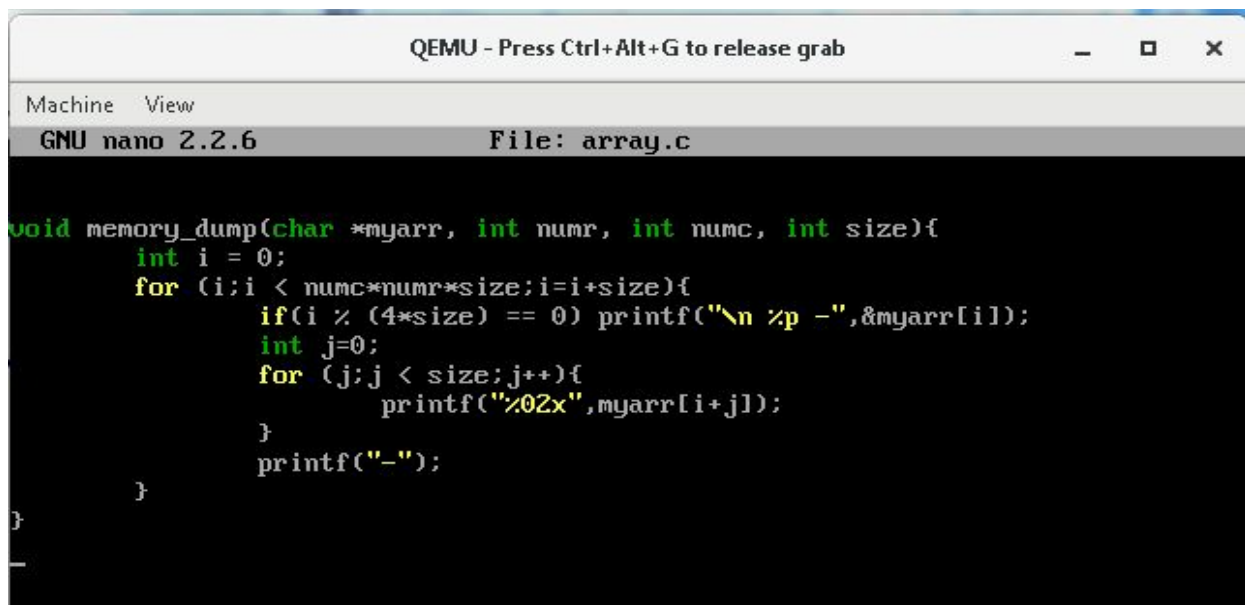
Code snippet 6

Task 2 - Memory dump

“Implement a memory dump program which can read through a character array and prints out each word in hexadecimal notation”

Most of the things the task requires came with a straightforward solution. We printed in hexadecimal by using the specifier “%x” to convert the value given as argument. And to print address we used the specifier “%p”. But the most challenging thing in the task was the requirement to print the word represented in 8 hexadecimal numbers. And if we look at the ASCII to hex translations, each character will translate into two hex numbers. This means that the input string has the length of 4 characters which will be stored in our matrix. Also, we cannot convert a full string into hex with the specifier. So we also needed to go through each character in each string in every cell of our matrix. And we changed the “%x” specifier to “%02x” so it printed each character in 2 hex numbers (e.g instead of printing “1” we print “01”) so we could make sure we go 8 hexadecimal numbers for each string. In the code below we use “i” to go through each byte in our array, and “j” to go through each string. We use “size” to go between cells, as each cell has “size” amount of bytes.

The rest of the code is just for formatting to print 4 words per line and the address of the first byte in each line.



The screenshot shows a QEMU window titled "QEMU - Press Ctrl+Alt+G to release grab". Inside, a nano editor is open with the file "array.c". The code defines a function "memory_dump" that takes a character array "myarr", dimensions "numr" and "numc", and a "size". It iterates through the array in rows of "size" bytes, printing the address of each row and then the 8 hexadecimal digits of each byte in the row, separated by hyphens.

```
void memory_dump(char *myarr, int numr, int numc, int size){
    int i = 0;
    for (i; i < numc*numr*size; i=i+size){
        if(i % (4*size) == 0) printf("\n %p -",&myarr[i]);
        int j=0;
        for (j; j < size; j++){
            printf("%02x",myarr[i+j]);
        }
        printf("-");
    }
}
```

Output;

```
00E84C18 - 6865636b - 6865636b - 6865636b - 6865636b -
00E84C28 - 6865636b - 6865636b - 6865636b - 6865636b -
00E84C38 - 6865636b -
```


Task 3 - Linked List

“Implement a struct which holds an element from a linked list. The List should contain an integer value, as well as a pointer to the next element in the list. Use the 2D array implementation from Task 1 to allocate an array of ten linked list elements.”

We started by creating 2 structs; LinkedList and; Node. Node has 2 elements, an integer which stores the value and a pointer to the next node. LinkedList only has one element which holds a pointer to the node at the Head of the list.

```
typedef struct node {
    int data;
    struct node * next;
}Node;

typedef struct linkedlist {
    struct node * Head;
}LinkedList;
```

The next step was creating 10 nodes with incrementing values where each node points to its incrementing node, with the exception of the tenth node where it points to NULL. We also let the LinkedList structures Head pointer point towards the first node.

```
LinkedList* List = (struct linkedlist*) malloc(sizeof(struct linkedlist));
Node * one = (struct node*) malloc(sizeof(struct node)); List->Head = one;
Node * two = (struct node*) malloc(sizeof(struct node)); one->data = 1; one->next = two;
Node * three = (struct node*) malloc(sizeof(struct node)); two->data = 2; two->next = three;
Node * four = (struct node*) malloc(sizeof(struct node)); three->data = 3; three->next = four;
Node * five = (struct node*) malloc(sizeof(struct node)); four->data = 4; four->next = five;
Node * six = (struct node*) malloc(sizeof(struct node)); five->data = 5; five->next = six;
Node * seven = (struct node*) malloc(sizeof(struct node)); six->data = 6; six->next = seven;
Node * eight = (struct node*) malloc(sizeof(struct node)); seven->data = 7; seven->next = eight;
Node * nine = (struct node*) malloc(sizeof(struct node)); eight->data = 8; eight->next = nine;
Node * ten = (struct node*) malloc(sizeof(struct node)); nine->data = 9; nine->next = ten;
ten->data = 10; ten->next = NULL;
```

The third step was adding all of these into the 2D array. Which didn't require much modification. We changed the size to the size of the structure, which can be obtained from a library function "sizeof". Each node required 8 bytes of storage in the 2D array, 4 bytes for the integer and 4 bytes for the pointer to the next node.

```
void two_d_store_linkedlist(char *myarray, int size, Node * node, int row, int col, int numr, int numc) {
    int position = ((col)*numr + row)*size;
    myarray[position] = node->data;
    myarray[position + 4] = node->next;
}
```

```

two_d_store_linkedlist(myarray, structsize, one, 0, 0, rows, cols);
two_d_store_linkedlist(myarray, structsize, two, 1, 0, rows, cols);
two_d_store_linkedlist(myarray, structsize, three, 0, 1, rows, cols);
two_d_store_linkedlist(myarray, structsize, four, 1, 1, rows, cols);
two_d_store_linkedlist(myarray, structsize, five, 0, 2, rows, cols);
two_d_store_linkedlist(myarray, structsize, six, 1, 2, rows, cols);
two_d_store_linkedlist(myarray, structsize, seven, 0, 3, rows, cols);
two_d_store_linkedlist(myarray, structsize, eight, 1, 3, rows, cols);
two_d_store_linkedlist(myarray, structsize, nine, 0, 4, rows, cols);
two_d_store_linkedlist(myarray, structsize, ten, 1, 4, rows, cols);

```

The fourth step in the process was modifying the `memory_dump` function, which only needed to change 2 lines of code. Remove the for-loop which goes through each byte in the string to instead look at the entire cell of data. Also change the `"%02x"` specifier to `"%08x"` to accommodate for the removal of the for-loop

```

void memory_dump_linkedlist(char *myarray, int numr, int numc, int size) {
    int i = 0;
    printf("%p - ", &myarray[i]);
    int count = 0;
    while (i < numc*numr*size) {
        //for (int j = 0; j < size; j++) {}
        printf("%08x _ ", myarray[i]);
        printf("%08x _ ", myarray[i+4]);

        i = i + size;
        count++;
        printf(" - ");
        if (count == 4) {
            printf("\n");
            printf("%p - ", &myarray[i]);
            count = 0;
        }
    }
}

```

So when we run it we get this output;

```

01454C18 - 00000001 _ 00000030 _ 00000030 _ 00000002 _ 00000002 _ 00000068 _ 00000068 _ 00000003 _
01454C28 - 00000003 _ ffffffff _ ffffffff _ 00000004 _ 00000004 _ ffffffff _ ffffffff _ 00000005 _
01454C38 - 00000005 _ 00000010 _ 00000010 _ 00000006 _

```


Which seems correct as each second block of data has the same value as the following block of data which indicates the pointer pointing towards the next value. And the task ended with the question “...*identify where the fifth node in the list is located.*”. The fifth node is the first 2 blocks of data in the second row and it doesn't seem to give me a normal hexadecimal number.