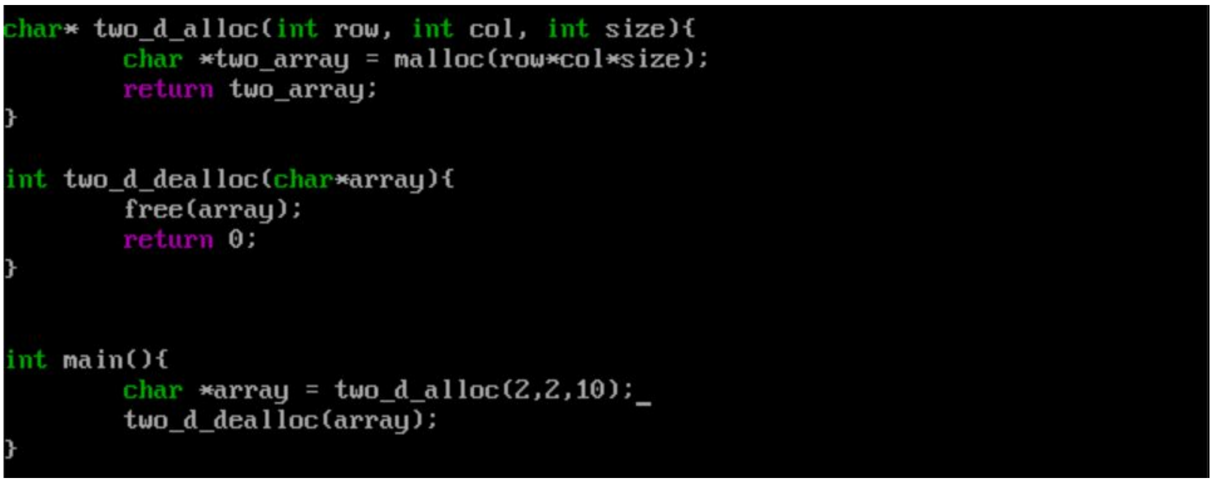
Lab3

Vide Hopper & Martin Karlsson

Task 1: Array storage

This lab was designed to use a one dimensional buffer to emulate a two dimensional array. We created two functions which allocated and deallocated the array. The allocation function was constructed in such way that it takes the row, column and the size of the element in order to allocate enough space for the array. The size of the element depends on the data type we want to work with. For example, the size of an integer is 4 bytes and the size an char is 1 byte.

When an allocation is performed the memory that has been allocated is on one line. We have created the alloc function in such a way that it returns a pointer to the first byte of the allocated memory. This means we can iterate through the memory.



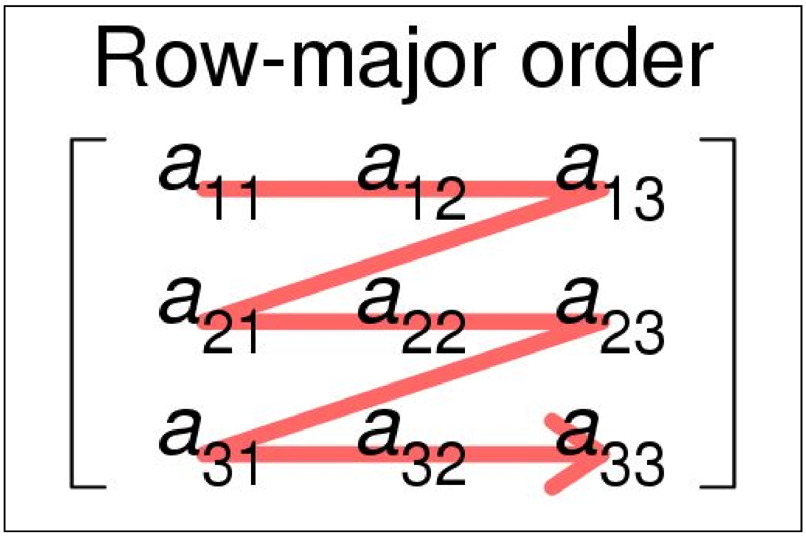
```
char* two_d_alloc(int row, int col, int size){
        char *two_array = malloc(row*col*size);
        return two_array;
}

int two_d_dealloc(char*array){
        free(array);
        return 0;
}


int main(){
        char *array = two_d_alloc(2,2,10);_
        two_d_dealloc(array);
}
```

Code 1, two_d_alloc & two_d_dealloc

The next step was to create a function that stores an element on the desired row and column. The function also needs the size of the matrix and the size of the element as arguments. In this task we were supposed to store an integer which means that the size of the element is 4 bytes. To store the value on the desired row and column we need to transform it into one dimensional translation of the given scenario.

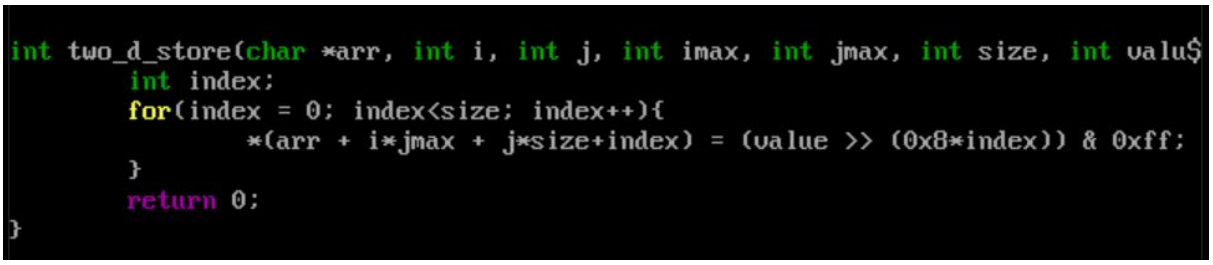

Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Picture 1, Row-major order 1D

Formula(1): *Index = Row*AmountColumn+Column*

In order to store the value at the right place which translates to the correct row and column we need to calculate its place on the one dimensional array. This translation can be done using *Formula(1)*. This works when the array starts at 0 and the matrix *Row* and *Column* also starts at 0. The *AmountColumn* is still the amount of columns, not one less. In the example above, to store a value at a21 or *Row*=1 & *Column*=0 the *AmountRows* is 3. This will result in Index=1*3+0=3.

Our implementation can be seen in *Code 2* where we take the first memory place of *arr* and then calculating the offset. The offset is calculated by taking the desired row times the max column size in order to make space for each column on every row then taking the column times the size, this is done so each element can fit in the respective slot in the array.



```
int two_d_store(char *arr, int i, int j, int imax, int jmax, int size, int valu$
        int index;
        for(index = 0; index<size; index++){
                *(arr + i*jmax + j*size+index) = (value >> (0x8*index)) & 0xff;
        }
        return 0;
}
```

Code 2, two_d_store

The for loop iterates through the 4 bytes of the integer and stores each byte in the associated memory place. For each iteration a right bit shift is done to the integer so a new byte can be stored. The *index* is the iterator and in this case it goes from 0-3, it determines by how many bytes the bit shift should be and also adds to the offset.

The first time the for loop runs the index is zero and no bit shift is done. This means that our one-byte buffer will store the most left byte of the integer.

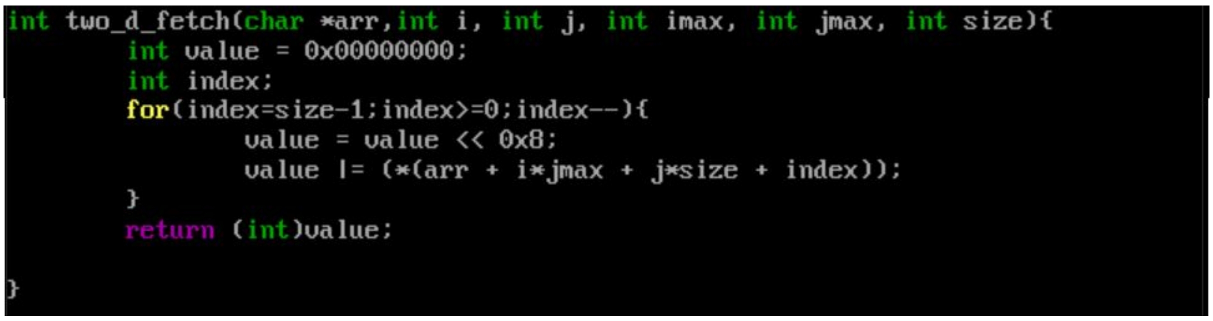
00001000 00000100 00000010 **00000001**

For the second iteration a one byte bit shift will be performed and the byte stored will be

00001000 00000100 **00000010**

When all iterations are done the whole integer is stored at the given row and column.

The next step is to create a function that can fetch a number from a given row and column and return that value.



```
int two_d_fetch(char *arr,int i, int j, int imax, int jmax, int size){
        int value = 0x00000000;
        int index;
        for(index=size-1;index>=0;index--){
                value = value << 0x8;
                value |= (*(arr + i*jmax + j*size + index));
        }
        return (int)value;

}
```

Code 3, two_d_fetch

As can be seen on *Code 3* the same principle is used as in the *two_d_store* function but with some modifications. The function needs the same arguments as *two_d_store* except that we don't need a value. Instead we create a variable called *value* that we assigned to be zero. Then the code proceed to the for loop that is modified so it starts with the max index which is 3 in this case (*size - 1*) and stops when index is 0.

The first iteration of the loop the value is left bit shifted by one. This does not mean anything to the current value which is

00000000 00000000 00000000 00000000

Then one byte from the array is added to the value with a or-operation. In the first iteration the offset of the array will give rightmost byte which in our example will be.

00000001 00000010 00000100 **00001000**

This will result in a new value for the variable *value*

00000000 00000000 00000000 00001000

In the second iteration a left bit shift is performed on the variable *value*

00000000 00000000 00001000 00000000

and a new byte from the array will be integrated to the value with the or-operation. In this iteration the byte is the second rightmost byte.

00000001 00000010 **00000100** 00001000

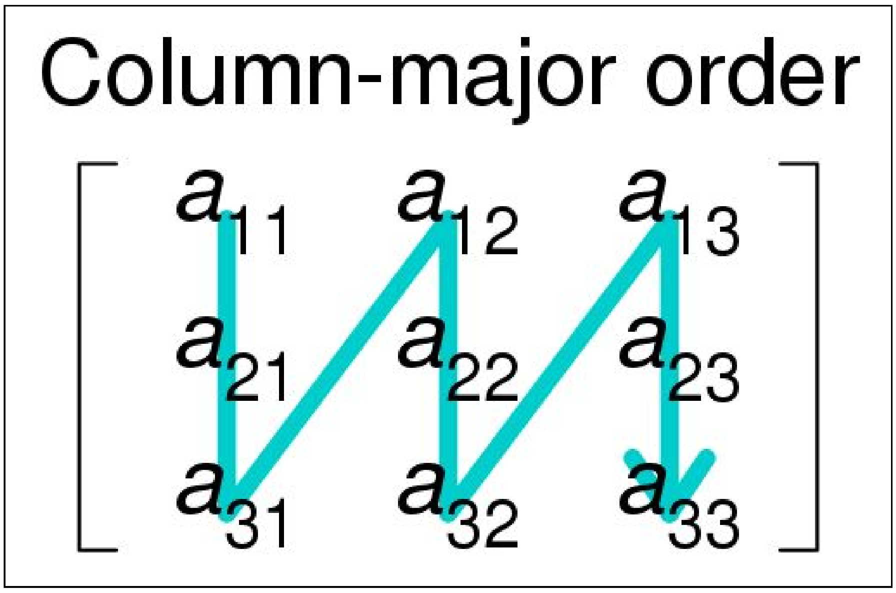
The variable *value* is now

00000000 00000000 00001000 00000100

When the loop is done *value* should have changed and should in this example be

00001000 00000100 00000010 00000001

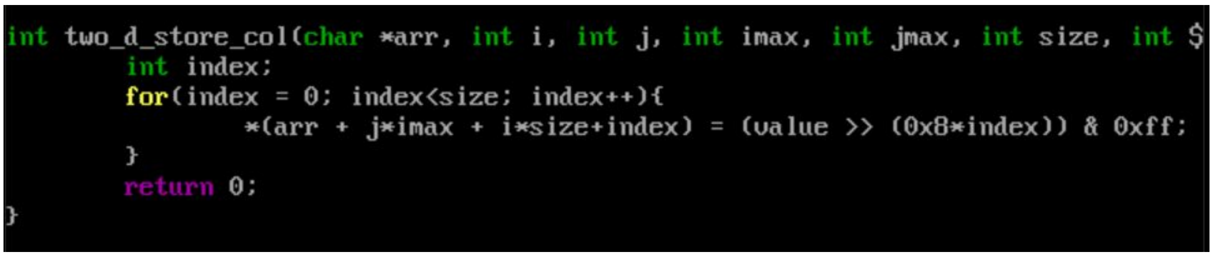
The last step it to return the variable as an int.


Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Picture 2, Column-major order

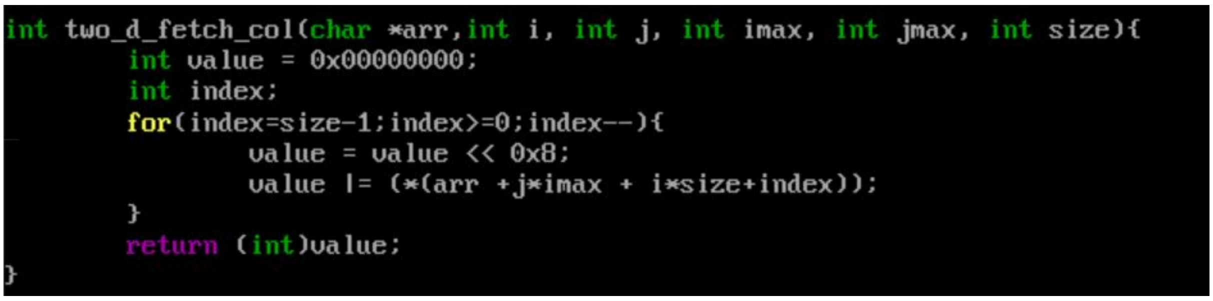
Formula(2): *Index = Column*AmountRow+Row*

We added a store and fetch function but using column major order instead. The difference between the original function is that column order uses *Formula(2)* instead of *Formula(1)*.



```
int two_d_store_col(char *arr, int i, int j, int imax, int jmax, int size, int $
        int index;
        for(index = 0; index<size; index++){
                *(arr + j*imax + i*size+index) = (value >> (0x8*index)) & 0xff;
        }
        return 0;
}
```

Code 4, two_d_store_col

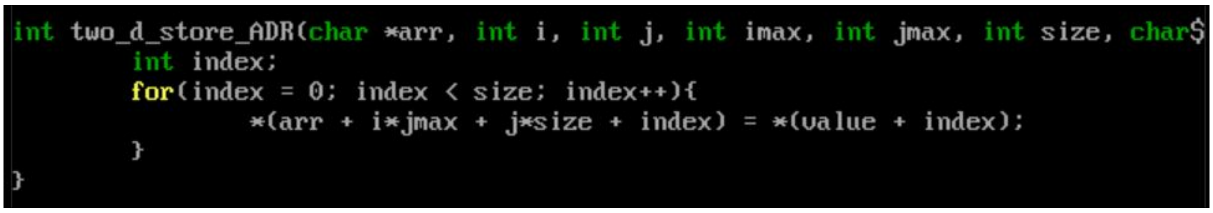


```
int two_d_fetch_col(char *arr,int i, int j, int imax, int jmax, int size){
        int value = 0x00000000;
        int index;
        for(index=size-1;index>=0;index--){
                value = value << 0x8;
                value |= (*(arr +j*imax + i*size+index));
        }
        return (int)value;
}
```

Code 5, two_d_fetch_col

The next functions we added was a store and fetch from the address of an datatype. These functions are made in such way that the data types can vary. The array needs to be allocated with a specific data type because since the values returned are only pointers to first memory byte. This memory byte then needs to be cast to the specified datatype.

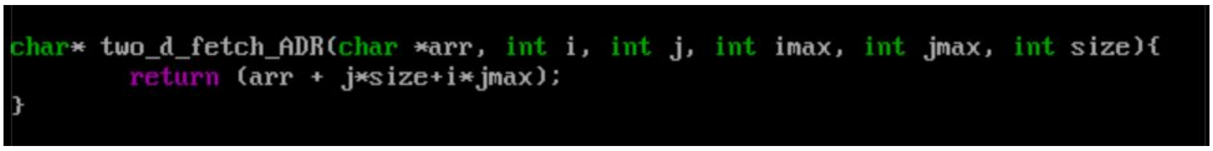
The store function was done by using a char pointer as an input argument. The pointer points to the first value of the data type. Then the loop will iterate through the two dimensional array and store each byte of the data type in the array.



```
int two_d_store_ADR(char *arr, int i, int j, int imax, int jmax, int size, char$
        int index;
        for(index = 0; index < size; index++){
                *(arr + i*jmax + j*size + index) = *(value + index);
        }
}
```

Code 6, two_d_store_ADR

The fetch function will only return the reference of the first byte of the value. This makes it useable if you want to store different sizes of data types. Outside this function you need to cast the pointer to its rightful data type.

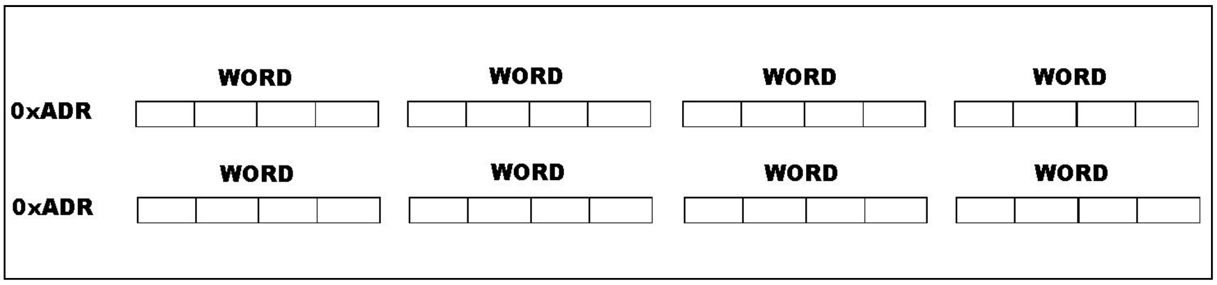


```
char* two_d_fetch_ADR(char *arr, int i, int j, int imax, int jmax, int size){
        return (arr + j*size+i*jmax);
}
```

Code 7, two_d_fetch_ADR

Task 2: Memory Dump

In this task we were supposed to create a “Memory Dump” function that reads through our array and prints each word in hexadecimal notation. A word is defined by 8 hexadecimal number. This means that each word contains 4 bytes and each row should contain 4 words which results in 16 bytes per line. Each row should start with the memory address of the first byte. *Picture 3* is a representation of the format of each memory dump.

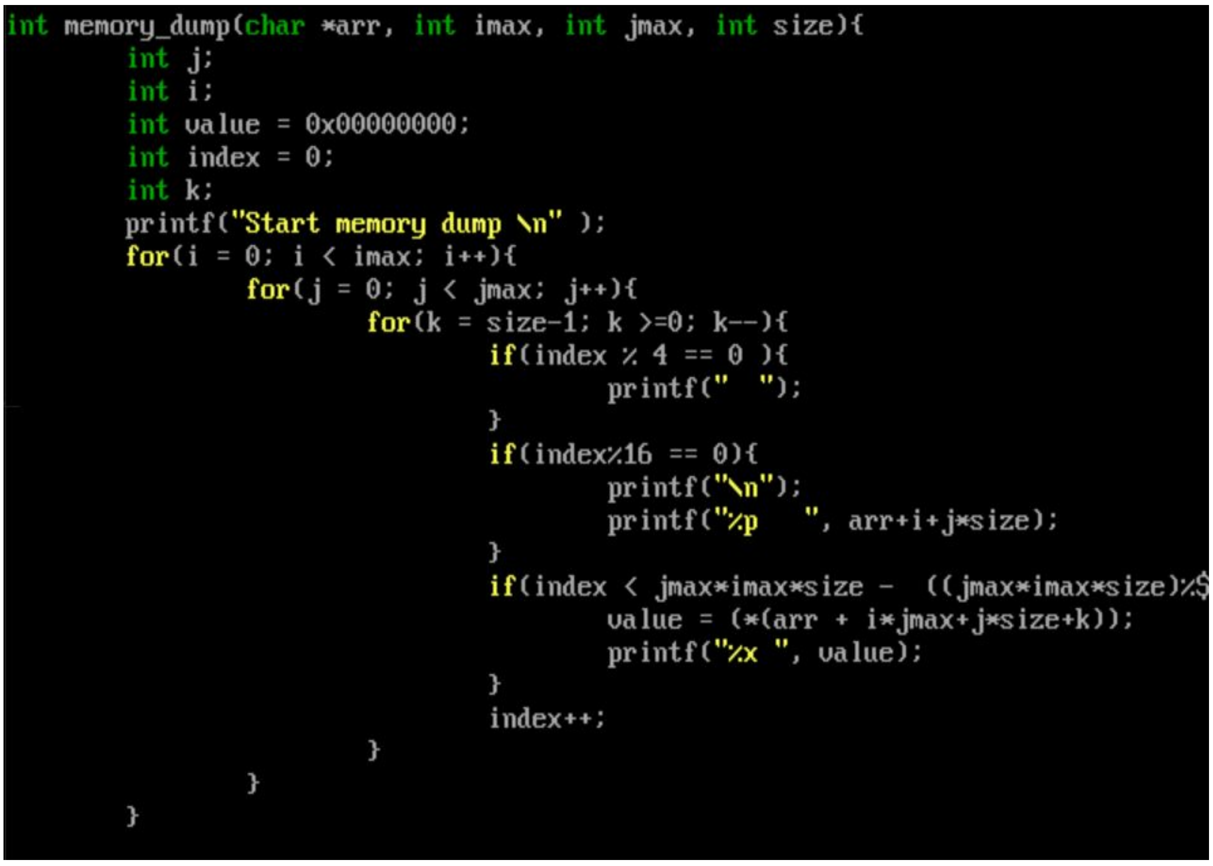


Picture 3, word representation

Our function, *Code 8*, takes the array, the size of the matrix and the size of the data types within as arguments. It then proceeds to a double nested for loop, the first loop iterates through the rows of the array and the other loop iterates through the columns. The last for loop iterates through all bytes in the array. A index is added to check which byte the program is currently on. Whenever the program has iterated over 4 bytes, it prints a space to mark the spacing between words. When it has iterated over 16 bytes, we need to change rows and print the address of the first byte. The last check is to make sure we only print the bytes if we can create a whole word with the bytes. If for example an array with only two bytes is taken in as input for this function it will not print those two bytes since they can not be represented in a word. This is done by calculating the amount of bytes in the array and then removing the bytes that will not fit in a word.

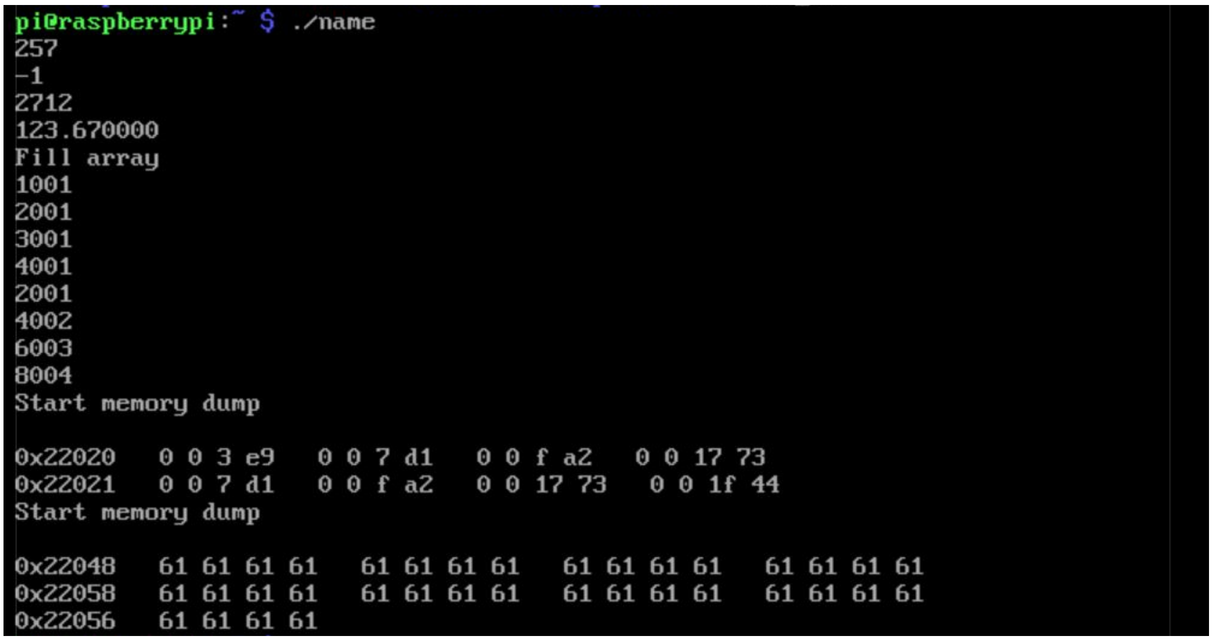
$$size - size \% 4$$

There is a scenario where memory dump only prints an address without any words attached to it. This happens when the elements in the array cannot build an entire word. This is a good notation that the array is not empty but not full enough to print a word.



```
int memory_dump(char *arr, int imax, int jmax, int size){
        int j;
        int i;
        int value = 0x00000000;
        int index = 0;
        int k;
        printf("Start memory dump \n" );
        for(i = 0; i < imax; i++){
                for(j = 0; j < jmax; j++){
                        for(k = size-1; k >=0; k--){
                                if(index % 4 == 0 ){
                                        printf("  ");
                                }
                                if(index%16 == 0){
                                        printf("\n");
                                        printf("%p    ", arr+i+j*size);
                                }
                                if(index < jmax*imax*size -  ((jmax*imax*size)%$
                                        value = (*(arr + i*jmax+j*size+k));
                                        printf("%x ", value);
                                }
                                index++;
                        }
                }
        }
```

Code 8, memory_dump



```
pi@raspberrypi:~ $ ./name
257
-1
2712
123.670000
Fill array
1001
2001
3001
4001
2001
4002
6003
8004
Start memory dump

0x22020   0 0 3 e9   0 0 7 d1   0 0 f a2   0 0 17 73
0x22021   0 0 7 d1   0 0 f a2   0 0 17 73   0 0 1f 44
Start memory dump

0x22048   61 61 61 61   61 61 61 61   61 61 61 61   61 61 61 61
0x22058   61 61 61 61   61 61 61 61   61 61 61 61   61 61 61 61
0x22056   61 61 61 61
```

Code 9, output from memory_dump

As seen on *Code 9*, our program outputs the information about a array in the same format that *Picture 3*. We choose to test *memory_dump* by using one array with integers and another with characters. Since an integer is 4 byte large, each element fits within each word. As for the character, each element is 1 byte of size. This means that one word should contain 4 characters. Our test case for characters was an array with 38 elements where each element was the character ‘a’. The character ‘a’ in hexadecimal notation is 61. Since the use of characters, every word should contain four elements. Also, the last word is not a full word since 38 is not a divideble with 4 so there should only be 36 elements displayed on *Code 9* which there is.

Conclusion

It’s unclear how to correctly represent a word. It is specified that a word is 4 bytes but it’s not specified if we were supposed to just convert it to hex byte by byte or if we should have directly translated 4 bytes into hexadecimal.

Binary	00001000	00000100	00000010	00000001	(1 byte wise)
Hexadecimal	8	4	2	1	

Binary	00001000000001000000001000000001	(4 byte wise)
Hexadecimal	8040201	

We choose to translate each byte to hexadecimal (first option) since we already had a 1 byte char pointer that we used to iterate through our matrix. For our solution this means that bytes that has at least four zeros at the beginning:

00000100

Will only be translated into *4* in hexadecimal and not *04* which could have made more sense depending on the use of the memory dump.

During the labs we had a hard time understanding the question and our labassistant didn’t completely understand the questions we asked. One question we asked was if we needed to bitshift and got the answer that we needed to do so. Because of this, we are unsure if the labs are done correctly but with this rapport we hope that the questions created for the labs are answered, from our point of view.