

# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya

### Task 1: Array Storage

This task is to implement store and fetch in a 2D byte array. By using 1D character buffer to emulate 2D array. The task also include 10 steps. Following the steps in the task we need to declare two functions `two_d_store` and `two_d_fetch` which we will modify depending on the steps from the tasks

Step 1. This to implement the two function mentioned above that allocates a `char*` to store an array of `NxM` element of a given size. The function have 4 arguments number of rows, number of columns, and size of each element.

#### Code:

For allocating memory we make the function `char *two_d_alloc(char *arr, int n, int m, int size)`

We allocate char pointer for size of int, and integer size of the row = n, and column = m.

```
GNU nano 2.2.6      File: array.c

#include <stdio.h>
#include <stdlib.h>
char *two_d_alloc(char *arr, int n, int m, int size){
    arr = malloc(size*n*m);
    printf("array address after alloce %p \n", arr);
    return arr;
}

void two_d_dealloc(char *arr){
    free(arr);
    printf("address after deallocate %p\n", arr);
}

void two_d_store(char *arr, int size, int n, int m, int npos, int mpos, int ele$
    int dest = npos*m + mpos;
    arr[dest] = element;
    arr[dest+1] = element >> 8;
    arr[dest+2] = element >> 16;
    arr[dest+3] = element >> 24;
    printf("address of value is:%p\n and the valuse are :%d,%d,%d,%d\n",
        &dest, arr[dest], arr[dest+1], arr[dest+2], arr[dest+3]);
}

int two_d_fetch(char *arr,int size, int n, int m, int npos, int mpos){
    int dest =(npos*m + mpos)*size;
    int fetch_value =

[ Read 39 lines ]
^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page  ^U UnCut Text^T To Spell
```

# Datorteknik

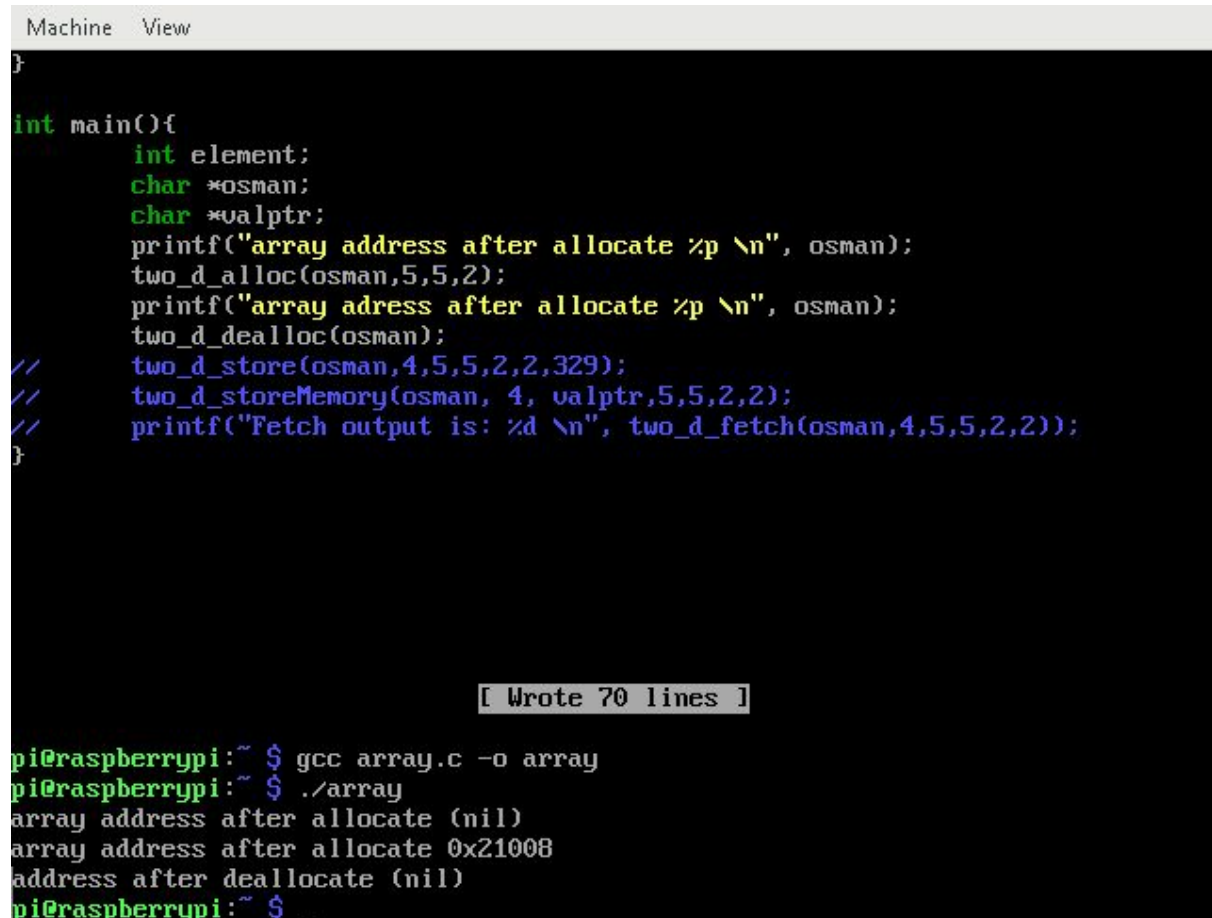
## Lab 3: Memory Organization

*Alen Gazibegovic*  
*Thanadon Suriya*

Step 2. Implement a function two\_d\_dealloc which deallocates two dimensional array stored in a character buffer.

**Code:** As can be seen in the code above, we create void function deallocate using free() function for pointer.

Step 3. Use the created functions above in main function.



```
Machine View
}

int main(){
    int element;
    char *osman;
    char *valptr;
    printf("array address after allocate %p \n", osman);
    two_d_alloc(osman,5,5,2);
    printf("array address after allocate %p \n", osman);
    two_d_dealloc(osman);
    // two_d_store(osman,4,5,5,2,2,329);
    // two_d_storeMemory(osman, 4, valptr,5,5,2,2);
    // printf("Fetch output is: %d \n", two_d_fetch(osman,4,5,5,2,2));
}

[ Wrote 70 lines ]

pi@raspberrypi:~ $ gcc array.c -o array
pi@raspberrypi:~ $ ./array
array address after allocate (nil)
array address after allocate 0x21008
address after deallocate (nil)
pi@raspberrypi:~ $
```

**Code:** Above is output and main function display of our code. We can see that the address is nil before allocation and nil after deallocation. In the arguments in main we define size of array (5x5) and int size for allocation in this example was 2 but later we used 4 for all.

Step 4. Implement a function two\_d\_store which stores an integer argument into a particular row and column of the array...

# Datorteknik

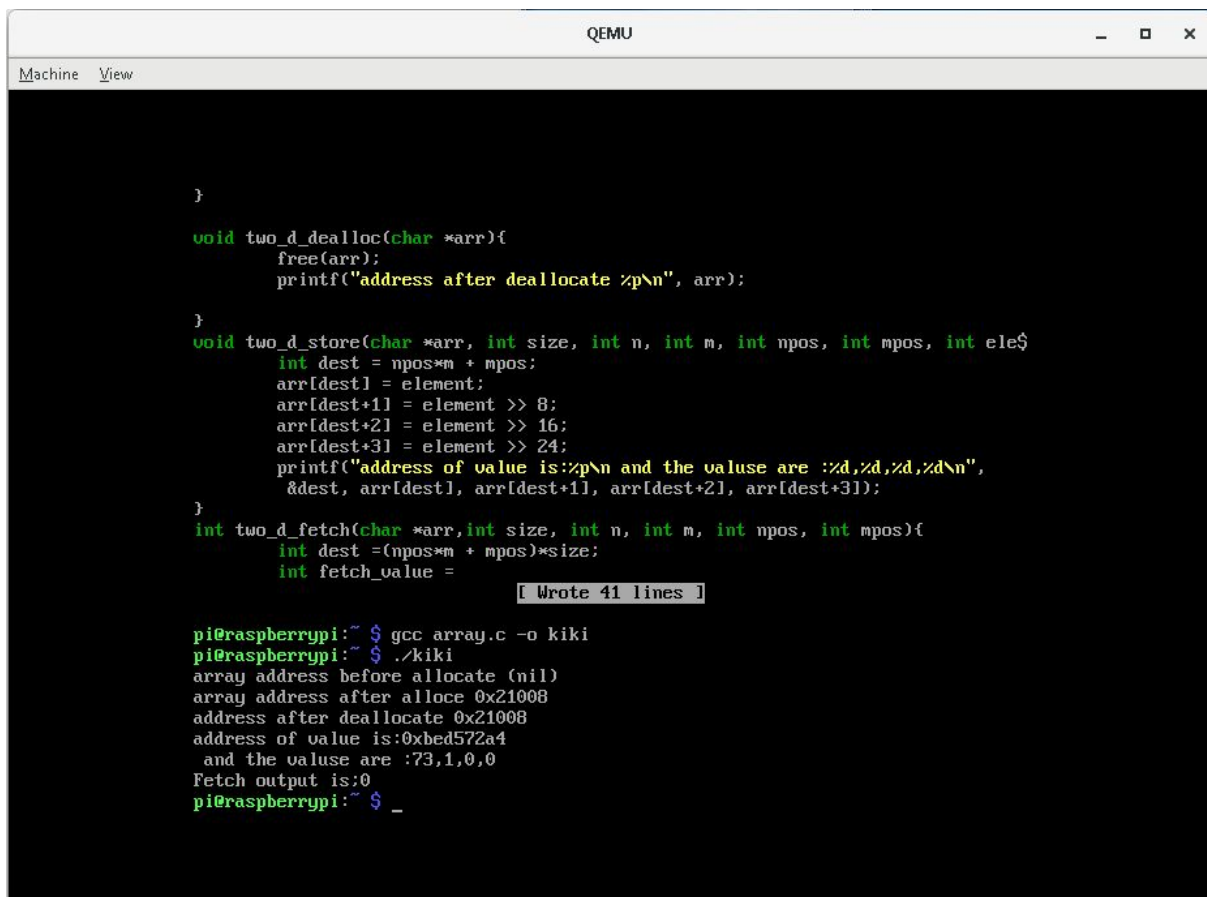
## Lab 3: Memory Organization

Alen Gazibegovic

Thanadon Suriya

**Code:** As can be seen in the picture below we have 7 arguments in the store function. The pointer, defining size for memory allocation, size of our 2D array  $n*m$  integer defined. `int npos` and `int mpos` refer to the position in which the last argument (element) is to be stored in our 2D array.

To calculate where the element is to be stored we use the formula for row-major positioning, desired row position \* column dimension of array + desired column position. Now we need to bitshift by 8 bits because if we do not do this, the largest element that we can store will be 255 ( $2^8 - 1$ ). Since an integer is 4 bytes this means we will have 4 sets of 8 bits. So in our code we must make 4 different positions of size 8 bits to store our full integer. Later its printed using `%d` syntax and accessing via array position `[dest+1/2/3/4]`. In output is displayed via a 1 for each 256 integer and rest in our 4 positions. So element 329 stored into our array will give output **73,1,0,0**.



```
Machine View

}

void two_d_dealloc(char *arr){
    free(arr);
    printf("address after deallocate %p\n", arr);
}

void two_d_store(char *arr, int size, int n, int m, int npos, int mpos, int ele$
    int dest = npos*m + mpos;
    arr[dest] = element;
    arr[dest+1] = element >> 8;
    arr[dest+2] = element >> 16;
    arr[dest+3] = element >> 24;
    printf("address of value is:%p\n and the valuse are :%d,%d,%d,%d\n",
        &dest, arr[dest], arr[dest+1], arr[dest+2], arr[dest+3]);
}

int two_d_fetch(char *arr,int size, int n, int m, int npos, int mpos){
    int dest =(npos*m + mpos)*size;
    int fetch_value =

I Wrote 41 lines

pi@raspberrypi:~ $ gcc array.c -o kiki
pi@raspberrypi:~ $ ./kiki
array address before allocate (nil)
array address after alloce 0x21008
address after deallocate 0x21008
address of value is:0xbcd572a4
and the valuse are :73,1,0,0
Fetch output is:0
pi@raspberrypi:~ $ _
```

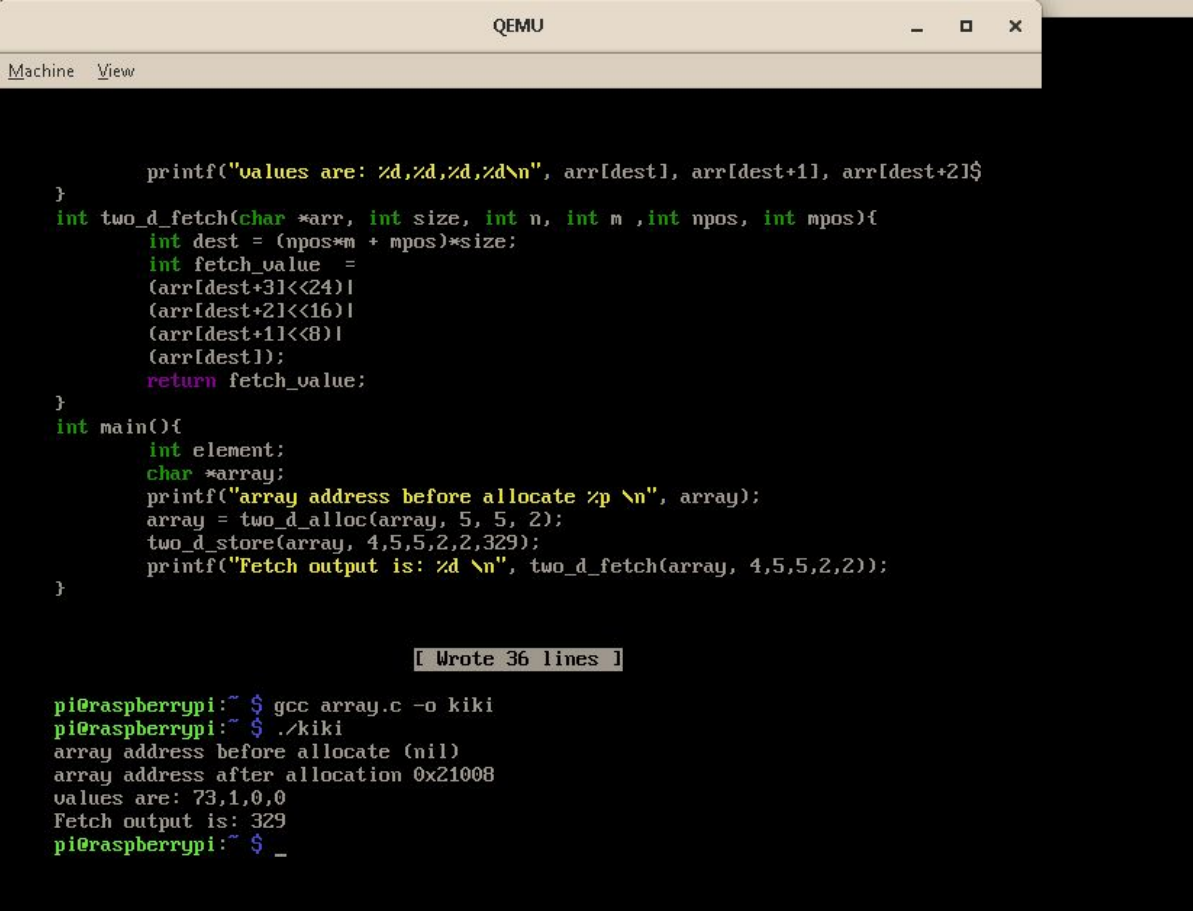
Step 5. • Implement a function `two_d_fetch` which returns the value of an integer argument stored in the memory array

# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya

**Code:** Below is shown the function `two_d_fetch` with 6 arguments now. It is one less than the store because here we need not specify which value we want to fetch, but access the value using the destination that it has been put in with our pointer. We define our dest variable (destination for the value that is input in `two_d_store` (element in our code)). After this we use same access to the full integer value by bitshift `<<` by 8. The output is the sum of the fetched 8-bit values, and as is seen below: 329 is returned from fetch value function.



```
Machine View

    printf("values are: %d,%d,%d,%d\n", arr[dest], arr[dest+1], arr[dest+2])$
}
int two_d_fetch(char *arr, int size, int n, int m, int npos, int mpos){
    int dest = (npos*m + mpos)*size;
    int fetch_value =
        (arr[dest+3]<<24)|
        (arr[dest+2]<<16)|
        (arr[dest+1]<<8)|
        (arr[dest]);
    return fetch_value;
}
int main(){
    int element;
    char *array;
    printf("array address before allocate %p \n", array);
    array = two_d_alloc(array, 5, 5, 2);
    two_d_store(array, 4,5,5,2,329);
    printf("Fetch output is: %d \n", two_d_fetch(array, 4,5,5,2,2));
}

[ Wrote 36 lines ]

pi@raspberrypi:~ $ gcc array.c -o kiki
pi@raspberrypi:~ $ ./kiki
array address before allocate (nil)
array address after allocation 0x21008
values are: 73,1,0,0
Fetch output is: 329
pi@raspberrypi:~ $ _
```

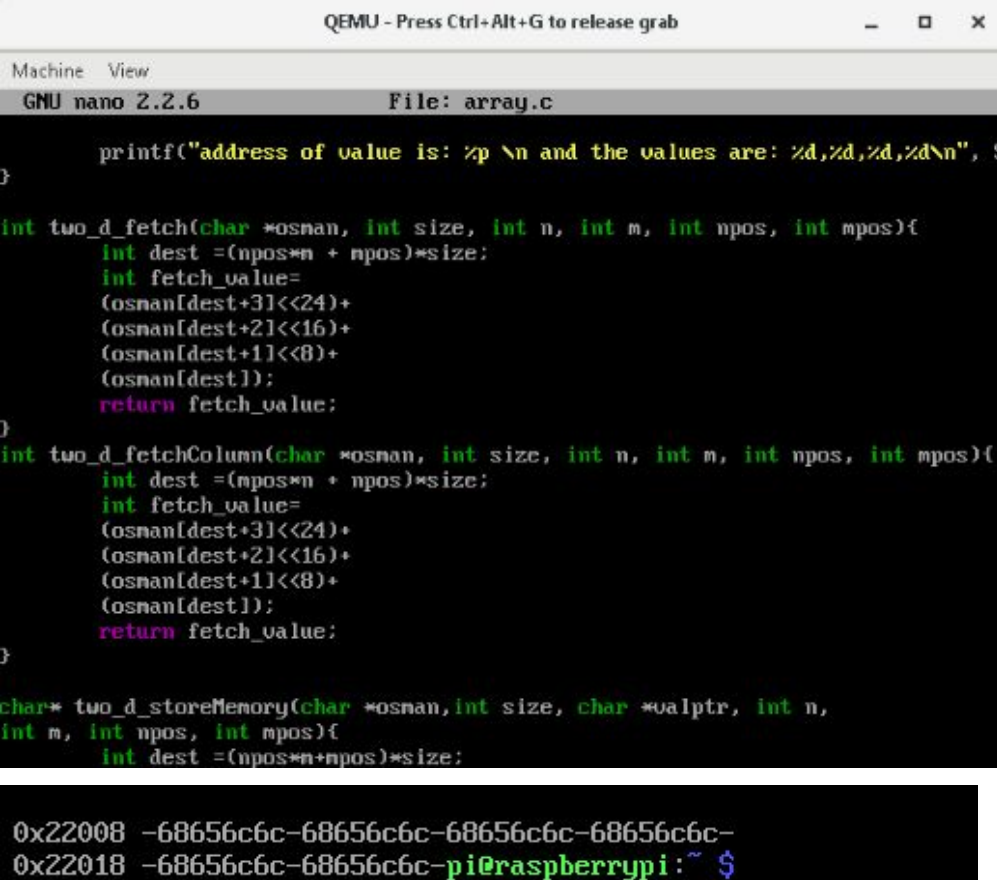
Step 6. Re-write `two_d_store` and `two_d_fetch` to use column-major format. Provide these as separate functions in your report. Test your implementations.

**Code:** The difference in implementation of column and major format is by calculating the destination of element to be stored in the array. In our code as previously shown we used `int dest = row position*columns + column position` for Row-Major format. For implementing Column-Major Format we use `column position * rows + row position`. So the code is exactly the same except in our file we write `int dest = mpos*n + npos`. **Below this on the next page is a screenshot of our `storeColumn` and `fetchColumn`, as can be seen there is no difference in the implementation except the one described above, which is a minor change in just one place.**

# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya



```
QEMU - Press Ctrl+Alt+G to release grab
Machine View
GNU nano 2.2.6 File: array.c

    printf("address of value is: %p \n and the values are: %d,%d,%d,%d\n", $
}

int two_d_fetch(char *osman, int size, int n, int m, int npos, int mpos){
    int dest =(npos*n + mpos)*size;
    int fetch_value=
    (osman[dest+3]<<24)+
    (osman[dest+2]<<16)+
    (osman[dest+1]<<8)+
    (osman[dest]);
    return fetch_value;
}

int two_d_fetchColumn(char *osman, int size, int n, int m, int npos, int mpos){
    int dest =(npos*n + mpos)*size;
    int fetch_value=
    (osman[dest+3]<<24)+
    (osman[dest+2]<<16)+
    (osman[dest+1]<<8)+
    (osman[dest]);
    return fetch_value;
}

char* two_d_storeMemory(char *osman,int size, char *valptr, int n,
int m, int npos, int mpos){
    int dest =(npos*n+mpos)*size;
```

```
0x22008 -68656c6c-68656c6c-68656c6c-68656c6c-
0x22018 -68656c6c-68656c6c-pi@raspberrypi:~ $
```

Step 7. Implement a version of two\_d\_store and two\_d\_fetch which uses the memory address of arguments. Instead of returning an integer, return the memory at which the integer is stored. You are free to choose between row-major or column-major storage.

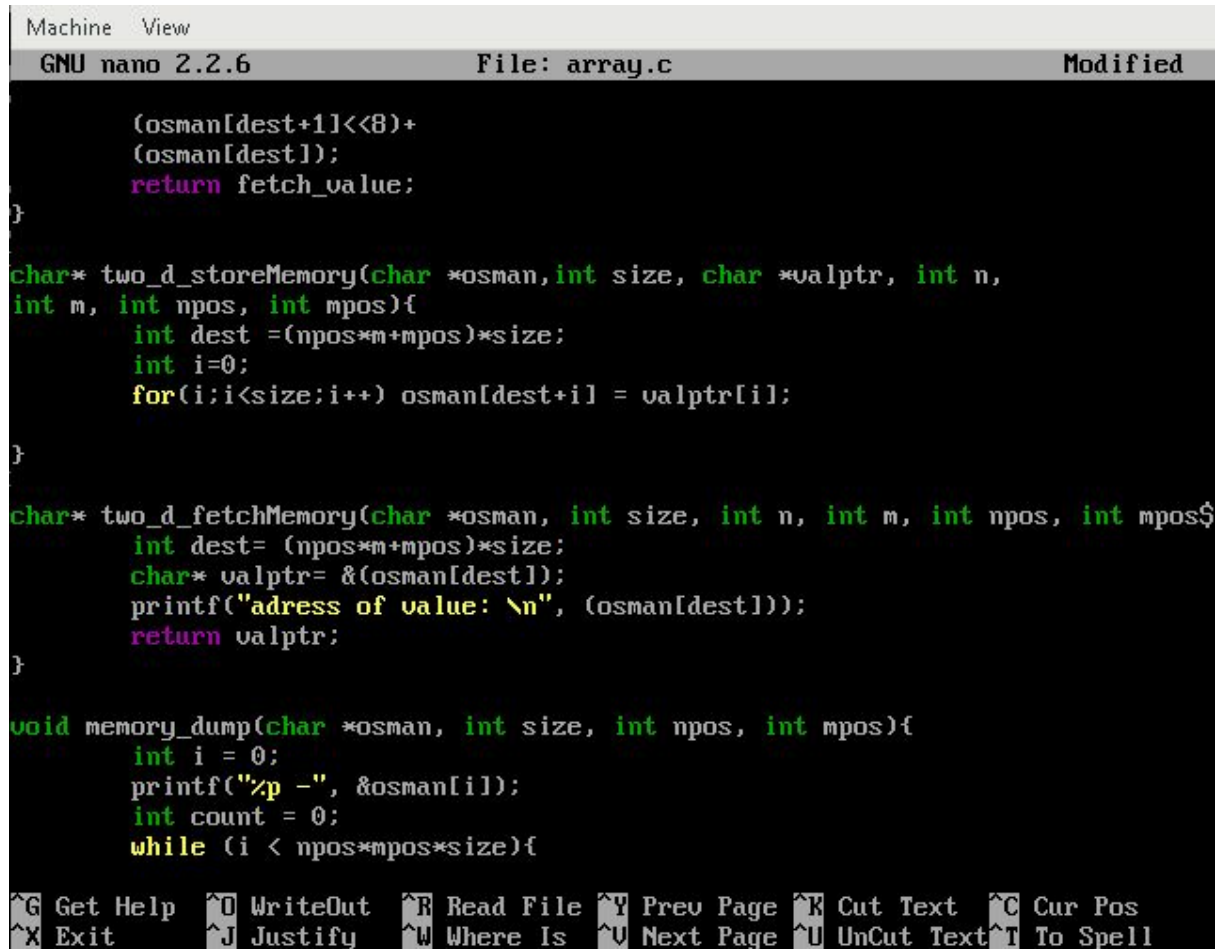
**Code:** For store memory address we will proceed from the row-major code as shown before. In StoreMemory and we need to use a character type valuepointer for accessing address, instead of as in two\_d\_store where we use int element variable to store into our array. This time around we decide to use for loop for indexing position in our array. Also we had re-written the code so our pointer type is now called osman instead of arr. The valptr is used as char because we do not access integer value in this task but rather memory address. Now

# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya

after indexing each position (4 of them) our valptr will return our addresses from array.



```
Machine View
GNU nano 2.2.6 File: array.c Modified

    (osman[dest+1]<<8)+
    (osman[dest]);
    return fetch_value;
}

char* two_d_storeMemory(char *osman,int size, char *valptr, int n,
int m, int npos, int mpos){
    int dest =(npos*m+mpos)*size;
    int i=0;
    for(i;i<size;i++) osman[dest+i] = valptr[i];
}

char* two_d_fetchMemory(char *osman, int size, int n, int m, int npos, int mpos){
    int dest= (npos*m+mpos)*size;
    char* valptr= &(osman[dest]);
    printf("adress of value: \n", (osman[dest]));
    return valptr;
}

void memory_dump(char *osman, int size, int npos, int mpos){
    int i = 0;
    printf("%p -", &osman[i]);
    int count = 0;
    while (i < npos*mpos*size){
        printf("%p -", &osman[i]);
        count++;
        if(count==4){
            printf("\n");
            count=0;
        }
        i++;
    }
}
```

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos  
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell

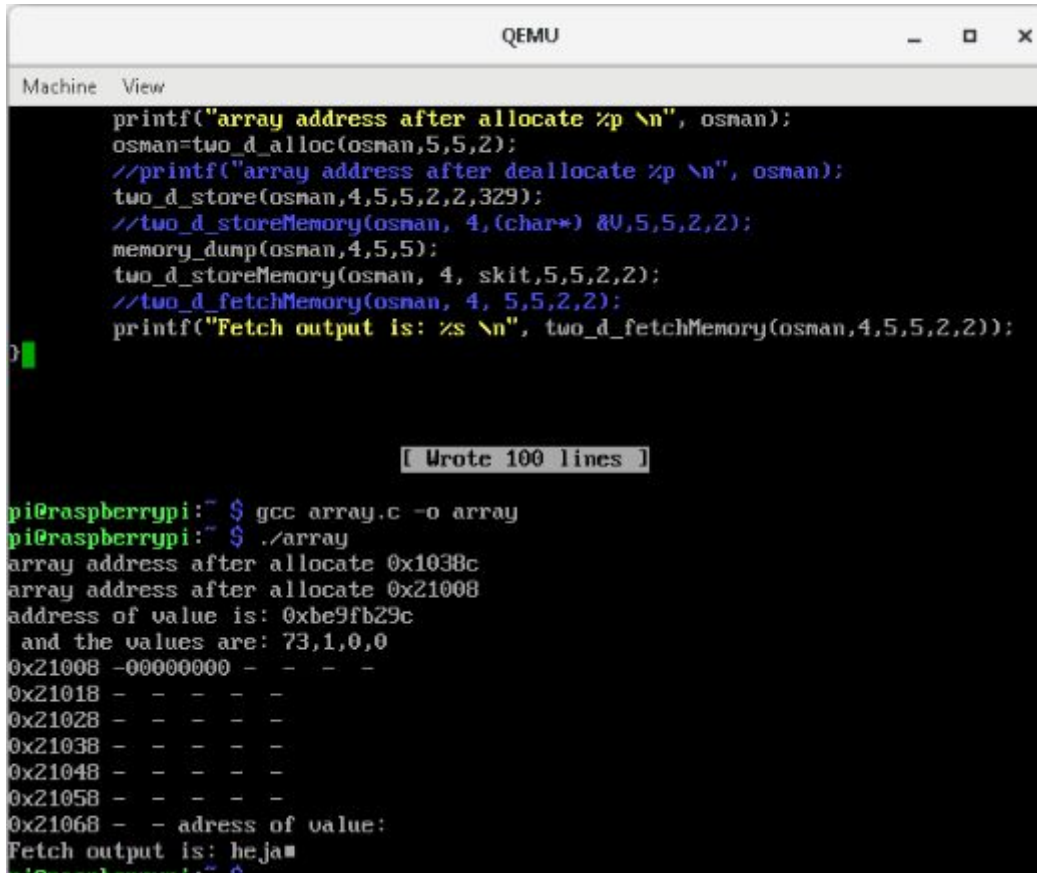
When looking at our code we can see that we are using char type pointer for accessing our new data type in the array. Because this is a char type pointer we can use casting to set the variable to the desired data type - for example, we can cast in the function declaration in main that valptr is: (int \*)&valptr, and therefore return the actual integer value. As can be seen in the examples below, we can use this functions for any data type.



# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya

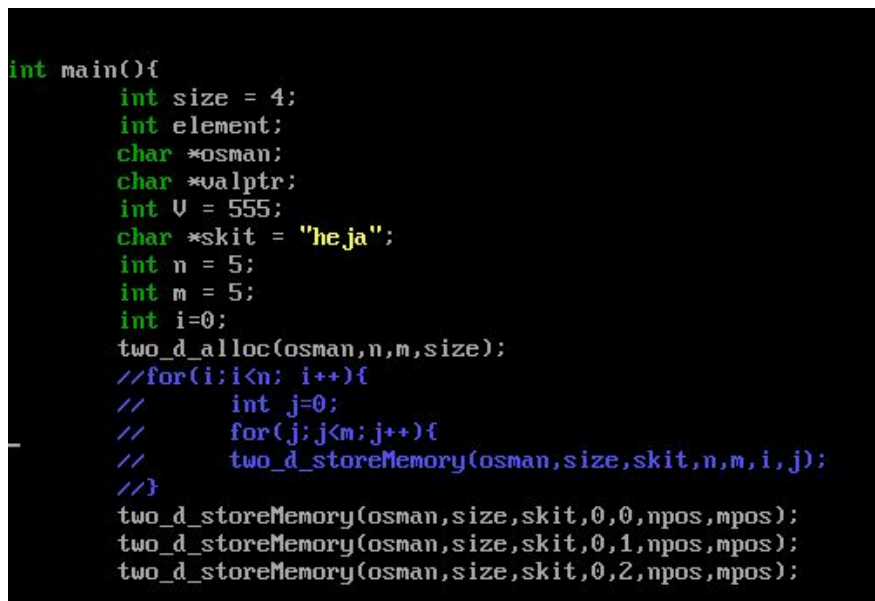


```
Machine View
printf("array address after allocate %p \n", osman);
osman=two_d_alloc(osman,5,5,2);
//printf("array address after deallocate %p \n", osman);
two_d_store(osman,4,5,5,2,2,329);
//two_d_storeMemory(osman, 4,(char*) &U,5,5,2,2);
memory_dump(osman,4,5,5);
two_d_storeMemory(osman, 4, skit,5,5,2,2);
//two_d_fetchMemory(osman, 4, 5,5,2,2);
printf("Fetch output is: %s \n", two_d_fetchMemory(osman,4,5,5,2,2));

[ Wrote 100 lines ]

pi@raspberrypi:~$ gcc array.c -o array
pi@raspberrypi:~$ ./array
array address after allocate 0x1038c
array address after allocate 0x21008
address of value is: 0xbe9fb29c
and the values are: 73,1,0,0
0x21008 -00000000 - - - -
0x21018 - - - - -
0x21028 - - - - -
0x21038 - - - - -
0x21048 - - - - -
0x21058 - - - - -
0x21068 - - address of value:
Fetch output is: heja
```

The output is Heja for the main input:



```
int main(){
    int size = 4;
    int element;
    char *osman;
    char *valptr;
    int U = 555;
    char *skit = "heja";
    int n = 5;
    int m = 5;
    int i=0;
    two_d_alloc(osman,n,m,size);
    //for(i;i<n; i++){
    //    int j=0;
    //    for(j;j<m;j++){
    //        two_d_storeMemory(osman,size,skit,n,m,i,j);
    //    }
    two_d_storeMemory(osman,size,skit,0,0,npos,mpos);
    two_d_storeMemory(osman,size,skit,0,1,npos,mpos);
    two_d_storeMemory(osman,size,skit,0,2,npos,mpos);
}
```

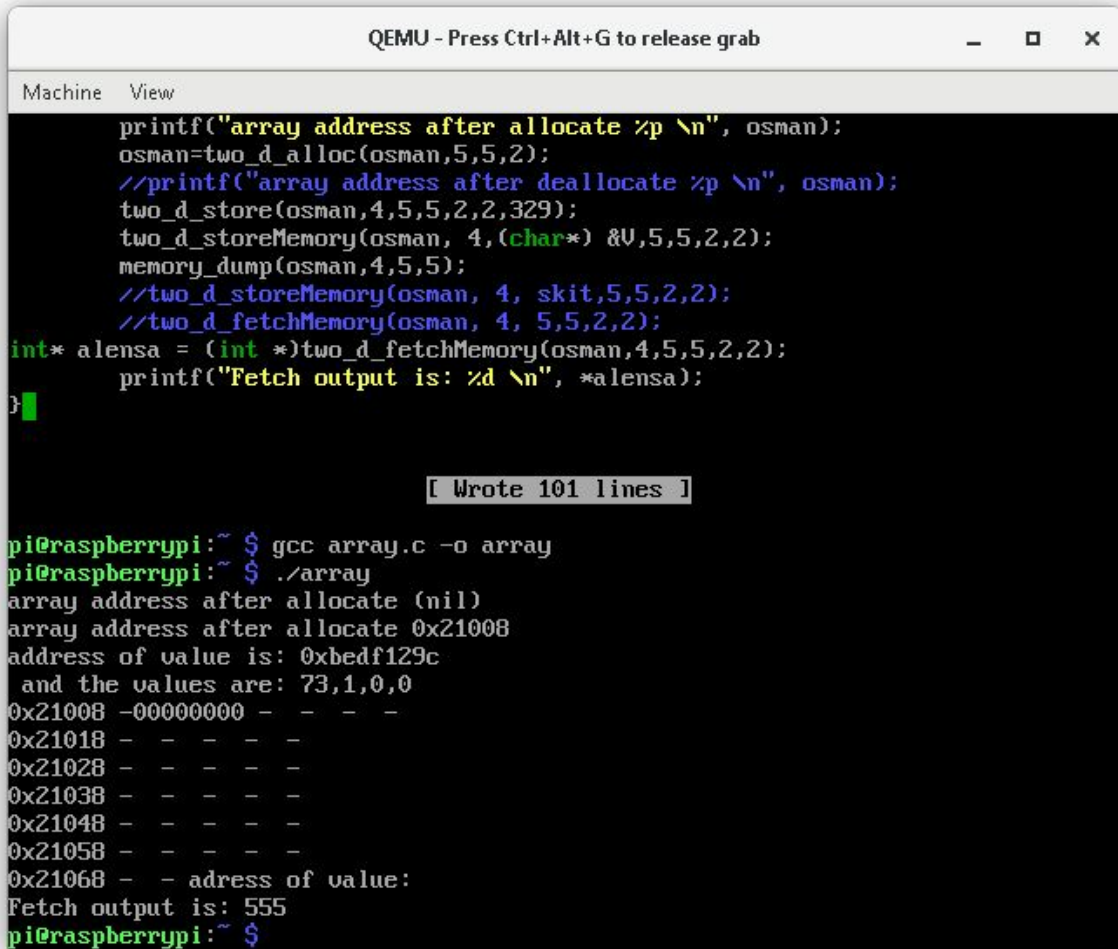
Output for integer type, where integer V is declared `int V = 555;` and cast with `int* ptrname = (int*)two_d_fetchM...(osman,4,5,5,2,2)` and this ptrnr returned as `%d` in

# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya

print function:



```
QEMU - Press Ctrl+Alt+G to release grab

Machine  View

printf("array address after allocate %p \n", osman);
osman=two_d_alloc(osman,5,5,2);
//printf("array address after deallocate %p \n", osman);
two_d_store(osman,4,5,5,2,2,329);
two_d_storeMemory(osman, 4,(char*) &U,5,5,2,2);
memory_dump(osman,4,5,5);
//two_d_storeMemory(osman, 4, skit,5,5,2,2);
//two_d_fetchMemory(osman, 4, 5,5,2,2);
int* alensa = (int *)two_d_fetchMemory(osman,4,5,5,2,2);
printf("Fetch output is: %d \n", *alensa);
}

[ Wrote 101 lines ]

pi@raspberrypi:~ $ gcc array.c -o array
pi@raspberrypi:~ $ ./array
array address after allocate (nil)
array address after allocate 0x21008
address of value is: 0xbedf129c
and the values are: 73,1,0,0
0x21008 -00000000 - - - -
0x21018 - - - - -
0x21028 - - - - -
0x21038 - - - - -
0x21048 - - - - -
0x21058 - - - - -
0x21068 - - adress of value:
Fetch output is: 555
pi@raspberrypi:~ $
```

### Task 2: Memory Dump

*Implement a memory dump program which can read through a character array and prints out the each word in hexadecimal notation. Separate words in groups of four words per line, where each word is represented by 8 hexadecimal numbers. Print the memory address of the first byte on every line. Test your memory dump on the character array you implemented in Task 1.*

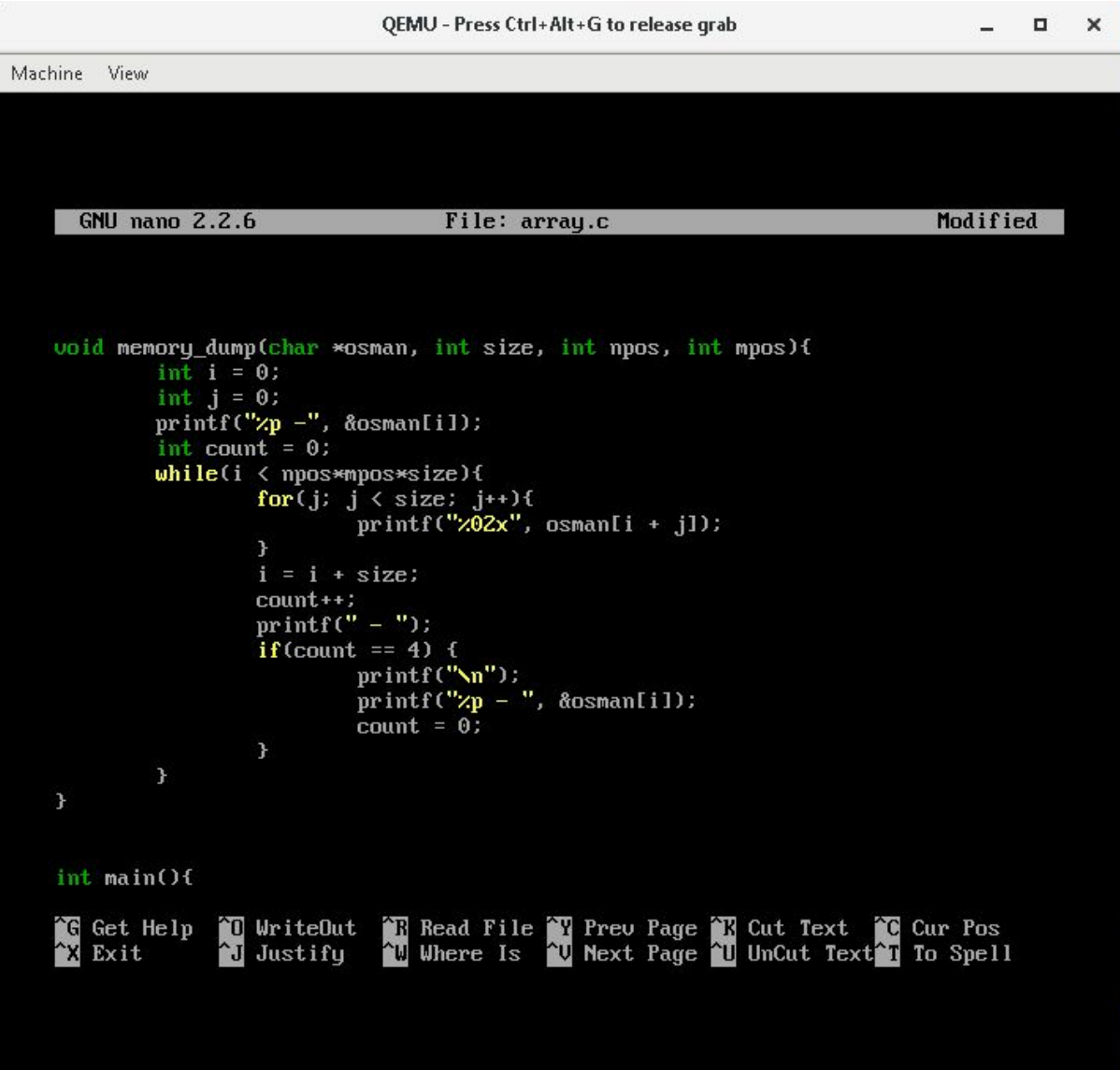


# Datorteknik

## Lab 3: Memory Organization

Alen Gazibegovic  
Thanadon Suriya

The function memory\_dump consists of the 4 arguments as seen in declaration of function. We initialize the loop i and j, variables and count variable, (used for measuring iteration of each loop to make each word 4 bits - in the code you can see that we identify when count == 4 for this), to 0 and print the address of first character therefore bit. The rest is pretty straight-forward and implementation could be handled without while but instead for loop but we use like this. The specifiers can be modified to different lengths of printing hexadecimal number which is required for accessing each character in string by 02x specifier. This will be printed size-amount of times because our dimension of the matrix declared in while condition.



```
QEMU - Press Ctrl+Alt+G to release grab

Machine View

GNU nano 2.2.6 File: array.c Modified

void memory_dump(char *osman, int size, int npos, int mpos){
    int i = 0;
    int j = 0;
    printf("%p -", &osman[i]);
    int count = 0;
    while(i < npos*mpos*size){
        for(j; j < size; j++){
            printf("%02x", osman[i + j]);
        }
        i = i + size;
        count++;
        printf(" - ");
        if(count == 4) {
            printf("\n");
            printf("%p - ", &osman[i]);
            count = 0;
        }
    }
}

int main(){

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

# Datorteknik

## Lab 3: Memory Organization

*Alen Gazibegovic*

*Thanadon Suriya*

The output shows the following addressing, the input word was 4-bits string “Heja” and we can see here the addressing of the input which became “Heja” 6 times. The address however seem to shift 10 bytes independent of how many repeats of the input we have. Then the address of each first byte is addressed. .

```
0x22008 -68656c6c-68656c6c-68656c6c-68656c6c-  
0x22018 -68656c6c-68656c6c-pi@raspberrypi:~ $
```