

Course

ELEC-A7100

Course materials

Your points

Microsoft Teams

Code Vault

This course has already ended.

« 1 More structured data types

Course materials

3 Command line arguments »

ELEC-A7100 / 7. Multidimensional Arrays / 2 Multidimensional arrays

Seventh round tasks as a **ZIP file**.

This section is mainly for **multidimensional arrays**, but before dealing with multidimensional arrays, a brief introduction is given to **Enumerated data types**, which are used to improve readability of the program.

Multidimensional arrays is not a new feature of the C language, but they are arrays, which consist of arrays. Arrays can be built either statically or dynamically. Especially in the latter case, you must be careful with memory allocations, use and releasing memory.

Multidimensional arrays

Static arrays

C supports also multidimensional arrays. In general, any number of array dimensions is possible, but here we focus just on two-dimensional arrays for clarity.

Below is an example of a simple 3x3 static array and how it can be used.

```
1 #include <stdio.h>
2
3 int main(void) {
4     int matrix[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
5     int i,j;
6
7     // Print the matrix in rectangular format
8     for (j = 0; j < 3; j++) {
9         for (i = 0; i < 3; i++) {
10             printf("%d ", matrix[j][i]);
11         }
12         printf("\n");
13     }
14
15     return 0;
16 }
```

Line 4 in the main function defines variable **matrix** that is a two-dimensional 3x3 array. It is also initialized at the same time: note the initialization list notation of nested arrays for each row of the table. The format of the initialization list also illustrates what multidimensional arrays really are in C: arrays made of arrays.

After the initialization, the array is just printed on the screen as two-dimensional rectangle. The array is indexed similarly to one-dimensional array, but now there are two indexes in square brackets.


The diagram below illustrates how C implements a static two-dimensional array as the above. There are three arrays of integer arrays that have three integers each. In this case, all 9 elements are located in adjacent locations in memory.

 ./_images/array-2d-static.jpg

Passing a static array as a function parameter has a special notation, as shown below. With a static two-dimensional array, the function parameter declaration must include the number of columns, so that the function can interpret the array data type correctly. The number of "first-dimension" members can be omitted: as with one-dimensional arrays, **int arr[]** is equal to **int *arr**, i.e., it can be handled as a pointer. The below example modifies the above program by moving the array printing part into a dedicated function.

```
1 #include <stdio.h>
2
3 void printArray(int arr[][3]) {
4     int i,j;
5     // Print the matrix in rectangular format
6     for (j = 0; j < 3; j++) {
7         for (i = 0; i < 3; i++) {
8             printf("%d ", arr[j][i]);
9         }
10        printf("\n");
11    }
12 }
13
14 int main(void) {
15     int matrix[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
16
17     printArray(matrix);
18
19     return 0;
20 }
```

A two-dimensional array can also be declared as an array of pointers (**arr_p** in below example), as shown below. This allows more flexible function interface, that will work with different sizes of arrays. Each pointer in the array points to the first member of a one-dimensional array.

 ./_images/array-2d-static-dyn.jpg

```
1 #include <stdio.h>
2
3 void printArray(int **arr, int xs, int ys) {
4     int i,j;
5     // Print the matrix in rectangular format
6     for (j = 0; j < ys; j++) {
7         for (i = 0; i < xs; i++) {
8             printf("%d ", arr[j][i]);
9         }
10        printf("\n");
11    }
12 }
13
14 int main(void) {
15     int matrix[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
16     int *arr_p[3];
17
18     for (int i = 0; i < 3; i++) {
19         arr_p[i] = matrix[i];
20     }
21
22     printArray(arr_p, 3, 3);
23
24     return 0;
25 }
```

The elements in **arr_p** are of different data type than the first-degree elements of the "matrix" array: **sizeof(arr_p[0])** returns the length of one **int * pointer** (8 bytes for 64-bit address), but **sizeof(matrix[0])** is the length of an array consisting 3 integers (12 bytes in most systems). Therefore, the printArray() function above cannot be directly called using the matrix parameter. Fortunately C allows implicit translation of a one-dimensional integer array (matrix[i]) into a integer pointer to the first member of array (**arr_p[i]**), as done on line 19 above). After this operation calling **printArray()** function is possible.

Because now **arr** is delivered as pointer of pointers, it accepts different dimensions of arrays. Therefore we added **two** new parameters to function interface, to indicate the actual dimensions of the array. Otherwise the function implementation might not know the dimensions.

Task 7.2: Static arrays

Define a two dimensional 6x6 static array with name **taulu**. Value stored in each array index must be the product of **row** and **column** index. For example,

taulu[0][0] contains the value 0, taulu[2][3] contains the value 6, and taulu[5][5] contains the value 25.

All values are integers.


You will get points if the expected output is printed:

```
0 0 0 0 0 0
0 1 2 3 4 5
0 2 4 6 8 10
0 3 6 9 12 15
0 4 8 12 16 20
0 5 10 15 20 25
```

Points **20 / 20** My submissions **1**

Deadline Friday, 23 July 2021, 19:59

To be submitted alone

 This course has been archived (Saturday, 31 December 2022, 20:00).

Static arrays

Select your files for grading

 **main.c**

Choose File

No file chosen

Submit

Dynamic allocated multi-dimensional arrays

Similar to one-dimensional arrays, memory for multi-dimensional arrays can also be allocated dynamically from **heap** using (possibly multiple) malloc() calls (in comparison to single **malloc()** call). A multidimensional array could be allocated in different ways, but here we focus on a straight-forward design: first, the high-level array is allocated for storing a number of columns (i.e., one-dimensional arrays). Then, each column of the array is allocated with a separate malloc() call. The latter malloc() calls are similar to allocating a one-dimensional array. The first array holds a series of pointers to the beginning of one-dimensional arrays, and the second-degree arrays store the actual members.

Here is an example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void printArray(int **arr, int xs, int ys) {
5     int i,j;
6     // Print the matrix in rectangular format
7     for (j = 0; j < ys; j++) {
8         for (i = 0; i < xs; i++) {
9             printf("%2d ", arr[j][i]);
10        }
11        printf("\n");
12    }
13 }
14
15 int main(void) {
16     int **arr_p;
17     int xdim = 4;
18     int ydim = 5;
19
20     // first allocate array that points to arrays of rows
21     // (notice the data type in sizeof operation)
22     arr_p = malloc(ydim * sizeof(int *));
23     if (!arr_p)
24         return -1; // memory allocation failed
25
26     for (int j = 0; j < ydim; j++) {
27         arr_p[j] = malloc(xdim * sizeof(int));
28         if (!arr_p[j]) {
29             // memory allocation failed, release memory
30             // will have to go through previously allocated rows
31             for (int i = 0; i < j; i++) {
32                 free(arr_p[i]);
33             }
34             free(arr_p);
35             return -1;
36         }
37         for (int i = 0; i < xdim; i++) {
38             // fill matrix with values, multiplication table
39             arr_p[j][i] = (i+1) * (j+1);
40         }
41     }
42
43     printArray(arr_p, xdim, ydim);
44
45     // release the memory
46     for (int j = 0; j < ydim; j++) {
47         free(arr_p[j]);
48     }
49     free(arr_p);
50     return 0;
51 }
```

Such an **array** would be located in memory somehow in the following way:

 ./_images/array-2d.jpg

This example continues modifying the previous simple matrix example. The **printArray()** function is same as before, but it is now called with two-dimensional array that is allocated from heap. Now the data type of arr_p variable is **int ****, a pointer to pointer(s), because with arrays a pointer points to the beginning of array.

The first degree array can be indexed using **arr_p[j]** (which is the same as ***(arr_p + j)**, as always). Because there is one deference, one "star" of the data type goes away. Therefore the data type of **arr_p[j]** is **int ***, i.e., a one-dimensional array. This is why on line 22 the number of rows is multiplied by **sizeof(int *)** in a malloc call: **int *** is the type of single member in the first-degree array.

Each of the **arr_p[j]** arrays can be indexed again with **arr_p[j][i]** (which is equivalent with ***(arr_p + j + i)**). There is another deference, and one "star" of the data type goes away. Therefore **arr_p[j][i]** is of type **int**.

Variables **ydim** and **xdim** hold the array dimensions. In this variant, the first-degree dimension is represented by ydim, as for vertical lines, and second-degree is for horizontal dimensions, but this could be also the other way around.

The same principle can be generalized into any number of dimensions: there could be three-dimensional arrays or four-dimensional arrays. The data types would just have more "stars" involved.

Task 7.3: Dynamic multidimensional array

In this task, you are supposed to implement two functions that are responsible of the memory allocation and freeing for a multidimensional array.

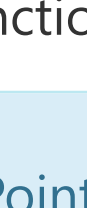
Function **char** allocate_memory(int* xdim, int ydim)** creates a multidimensional array variable that will contain *char* type elements and allocates memory for it. The program will expect that memory is allocated first for pointers according to parameter *ydim*. After that, for each pointer, memory should be allocated according to elements in parameter *xdim*. This means that the first pointer should have enough memory allocated to contain a string of length *xdim[0]* and so on. And since we are dealing with strings, you should take into consideration the terminating nil. Finally, return the created array.

Function **void free_array(char** w, int ydim)** will free all the memory allocated for the multidimensional array w.

Points **20 / 20** My submissions **1**


Deadline Friday, 23 July 2021, 19:59

To be submitted alone

 This course has been archived (Saturday, 31 December 2022, 20:00).

Dynamic multidimensional arrays

Select your files for grading

 **main.c**

Choose File

No file chosen

Submit

Arrays of strings

A common case for array of pointers is an array of strings. Essentially, this is a two-dimensional array as well, because strings are arrays of characters. In this case the different columns can have different lengths.

The following code represents the names of the months as an array:

```
1 #include <stdio.h>
2
3 int main(void) {
4     const char *months[] = { "January", "February", "March", "April", "May",
5                               "June", "July", "August", "September", "October",
6                               "November", "December" };
7
8     for (int i = 0; i < 12; i++) {
9         printf("%s\n", months[i]);
10    }
11    return 0;
12 }
```

In the above example, strings in the array are constant strings: they cannot be modified, but they can be accessed as normal array members. A single character of the months array could be accessed using **months[j][i]**, where **i** indicates the character in month string **j**. Because **printf()** format specifier **%s** on above line 9 assumes char * type, only the first-degree index is needed.

The strings can also be defined as modifiable strings in a similar way as in the basic case. Then the definition of months array, could be, for example:

```
char months[12][20] = { "January", "February", "March", "April", "May",
                        "June", "July", "August", "September", "October",
                        "November", "December" };
```

The above definition declares months as an array of 12 strings, and for each string 20 bytes are reserved. We could omit 12 in the definition, and just use **char months[][20]**, because the compiler will see the needed array length from the initialization list, but the size of each string must be given, so that the compiler can build the array appropriately. The strings will be located on consecutive memory locations, 20 bytes each.