

CH-230-A

Programming in C and C++

C/C++

Tutorial 4

Dr. Kinga Lipskoch

Fall 2020

Pointers and Arrays

- ▶ Ex: `char array[5];`
`char *array_ptr1 = &array[0];`
`char *array_ptr2 = array;`
`// the same as above`
- ▶ C allows pointer arithmetic:
 - ▶ Addition
 - ▶ Subtraction
- ▶ `*array_ptr` equivalent to `array[0]`
- ▶ `*(array_ptr+1)` equivalent to `array[1]`
- ▶ `*(array_ptr+2)` equivalent to `array[2]`
- ▶ What is `(*array_ptr)+1`?

Locating a Matrix Element in the Memory

- ▶ Consider the following
`int table[ROW][COL];`
where ROW and COL are constants
- ▶ `table` holds the address of the pointer to the first element
- ▶ `*table` holds the address of the first element
- ▶ What is the address of `table[i][j]`?
 $*(table + (i * COL + j))$
- ▶ One can determine the formula for an arbitrary multidimensional array with a similar pattern to the one above

Pointer Arithmetic with Arrays

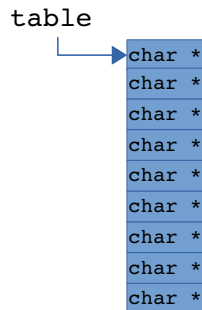
```
1 #include <stdio.h>
2 #define ROW 2
3 #define COL 3
4 int main() {
5     int arr[ROW][COL] = { {1, 2, 3}, {11, 12, 13} };
6     int i = 1;
7     int j = 2;
8     int* p = (int*) arr;           // needs explicit cast
9     printf("Address of [1][2]: %p\n", &arr[1][2]);
10    printf("Address of [1][2]: %p\n", p + (i * COL + j));
11    printf("Value of [1][2]: %d\n", arr[1][2]);
12    printf("Value of [1][2]: %d\n", *(p + (i * COL + j)));
13    printf("\n");
14    printf("Address of [0][0]: %p\n", p + (0 * COL + 0));
15    printf("Address of [0][1]: %p\n", p + (0 * COL + 1));
16    printf("Address of [0][2]: %p\n", p + (0 * COL + 2));
17    printf("Address of [1][0]: %p\n", p + (1 * COL + 0));
18    printf("Address of [1][1]: %p\n", p + (1 * COL + 1));
19    printf("Address of [1][2]: %p\n", p + (1 * COL + 2));
20    return 0;
21 }
```

Variably Sized Multidimensional Arrays

- ▶ Unidimensional arrays can be allocated "on the fly" using the `malloc()` function
- ▶ Possible also for multidimensional arrays, but more tricky
- ▶ Underlying idea: a pointer can point to the first element of a sequence
- ▶ A pointer to a pointer can then point to the first element of a sequence of pointers
 - ▶ And each of those pointers can point to first element of a sequence

Pointers to Pointers for Multidimensional Arrays (1)

- ▶ Consider the following
`char **table;`
- ▶ We can make table to point to an array of pointers to `char`
`table = (char **) malloc(sizeof(char *)
* N);`
- ▶ Every element in the array of N rows is a `char*`



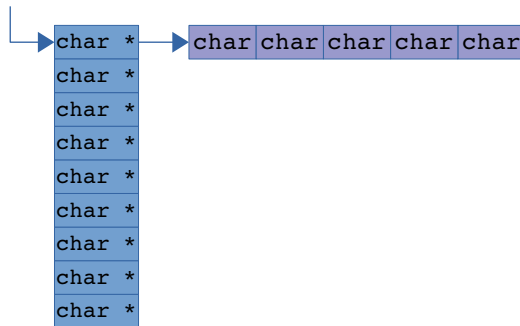
Pointers to Pointers for Multidimensional Arrays (2)

- ▶ Every pointer in the array can in turn point to an array
- ▶ In this way a two-dimensional array with N rows and M columns has been allocated

```
1 for (i = 0; i < N; i++)  
2   table[i] = (char *) malloc(sizeof(char) * M);
```

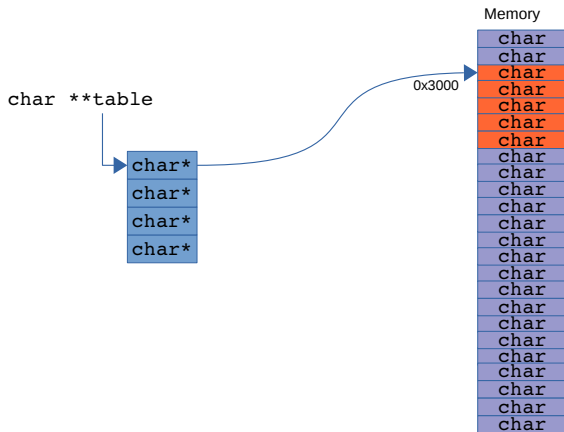
Pointers to Pointers for Multidimensional Arrays (3)

```
char **table
```



To access a generic element in the dynamically allocated matrix a matrix-like syntax can be used. Let us see why ...

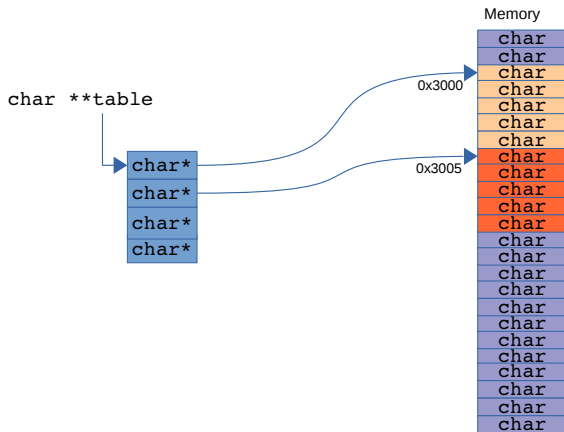
Allocating Space for a Multidimensional Array (1)



Case `i=0`

```
1 for (i = 0; i < 4; i++)  
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

Allocating Space for a Multidimensional Array (2)



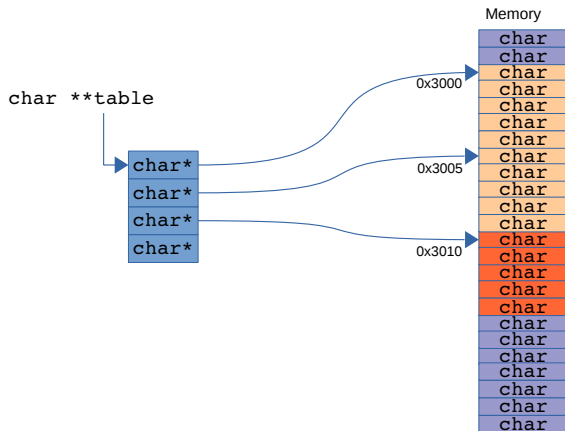
Case `i=1`

```

1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);

```

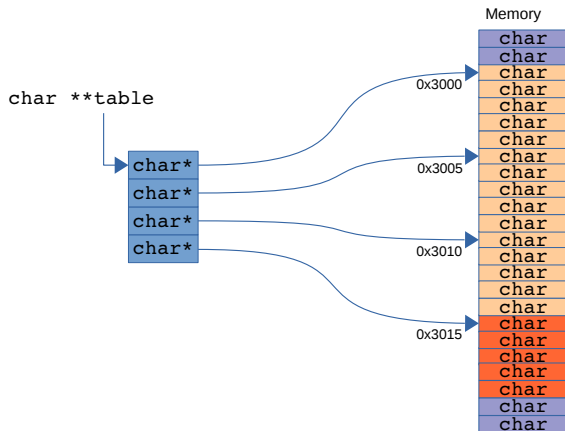
Allocating Space for a Multidimensional Array (3)



Case i=2

```
1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

Allocating Space for a Multidimensional Array (4)



Case i=3

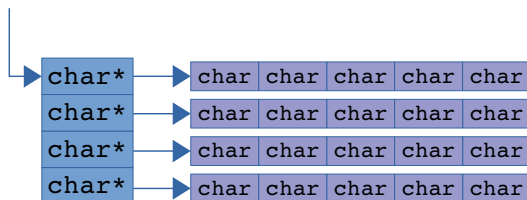
```

1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);

```

Drawing Memory in a Different Way: The Result is a Table

`char **table`



```
1 for (i = 0; i < 4; i++)  
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

De-allocating a Pointer to Pointer Structure

- ▶ Everything you have allocated via `malloc()` must be de-allocated via `free()`
- ▶ **Ex:** De-allocation of a 2D array with N elements

```
1 int i;  
2 for (i = 0; i < N; i++)  
3     free(table[i]);  
4 free(table);
```

Working with 2D Dynamic Arrays

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void set_all_elements(int **arr, int numrow, int numcol) {
4     int r, c;
5     for (r = 0; r < numrow; r++)
6         for (c = 0; c < numcol; c++)
7             arr[r][c] = r * c;    // some value ...
8 }
9 int main() {
10     int **table, row;
11     table = (int **) malloc(sizeof(int *) * 3);
12     if (table == NULL)
13         exit(1);
14     for (row = 0; row < 3; row++) {
15         table[row] = (int *) malloc(sizeof(int) * 4);
16         if (table[row] == NULL)
17             exit(1);
18     }
19     set_all_elements(table, 3, 4);
20 }
```

Static vs. Dynamic Array Allocation (1)

- ▶ `int a[n][m]` leads to an index offset calculation using the known array dimensions
- ▶ `int **a` treats `a` as an array `int *[]` and once indexed the result as an array of `int []`
- ▶ Statically allocated arrays occupy less memory
- ▶ Pointers to pointers allow tables where every row can have its own dimension
- ▶ One can have pointers to pointers to pointers (e.g., `int ***`) to have 3D data structures

Static vs. Dynamic Array Allocation (2)

- ▶ Static allocation
 - ▶ `int a[100][50]; int b[n][m];`
 - ▶ Syntax for allocation is easy
 - ▶ Release/reallocation not possible at runtime
 - ▶ Allocated memory is contiguous
- ▶ Dynamic allocation
 - ▶ `int **a; int *b[100], int ***c; ...`
 - ▶ Call(s) of `malloc` is needed
 - ▶ Syntax for allocation is more difficult
 - ▶ Release/reallocation possible at runtime using `free`, `realloc`
 - ▶ Allocated memory can be, but in general is not contiguous
- ▶ Passing arrays to functions: `static_dyn_allocation.c`
- ▶ Further reading/study:
<https://www.cse.msu.edu/~cse251/lecture11.pdf>