CH-230-A

Programming in C and C++

C/C++

Lecture 4

Dr. Kinga Lipskoch

Fall 2019

Pointers and Arrays

- Ex:
```c
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

```
#include <stdio.h>
#define ROW 2
#define COL 3
int main() {
  int arr[ROW][COL] = { {1, 2, 3}, {11, 12, 13} };
  int i = 1;
  int j = 2;
  int* p = (int*) arr;         // needs explicit cast
  printf("Address of [1][2]: %p\n", &arr[1][2]);
  printf("Address of [1][2]: %p\n", p + (i * COL + j));
  printf("Value of [1][2]: %d\n", arr[1][2]);
  printf("Value of [1][2]: %d\n", *(p + (i * COL + j)));
  printf("\n");
  printf("Address of [0][0]: %p\n", p + (0 * COL + 0));
  printf("Address of [0][1]: %p\n", p + (0 * COL + 1));
  printf("Address of [0][2]: %p\n", p + (0 * COL + 2));
  printf("Address of [1][0]: %p\n", p + (1 * COL + 0));
  printf("Address of [1][1]: %p\n", p + (1 * COL + 1));
  printf("Address of [1][2]: %p\n", p + (1 * COL + 2));
  return 0;
}
```

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)

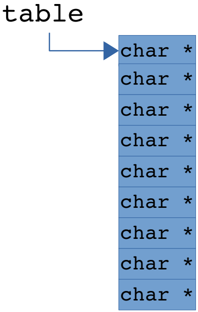
- Consider the following

  ```
  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*`

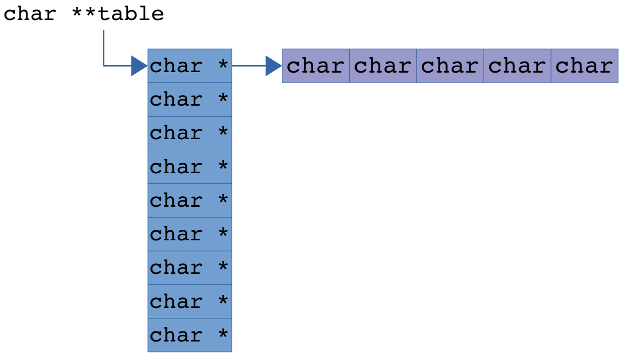


Pointers to Pointers for Multidimensional Arrays (2)

- 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

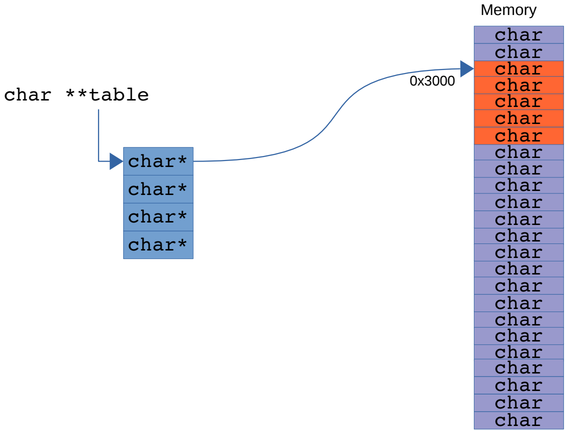
```c
for (i = 0; i < N; i++)
  table[i] = (char *) malloc(sizeof(char) * M);
```

Pointers to Pointers for Multidimensional Arrays (3)



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

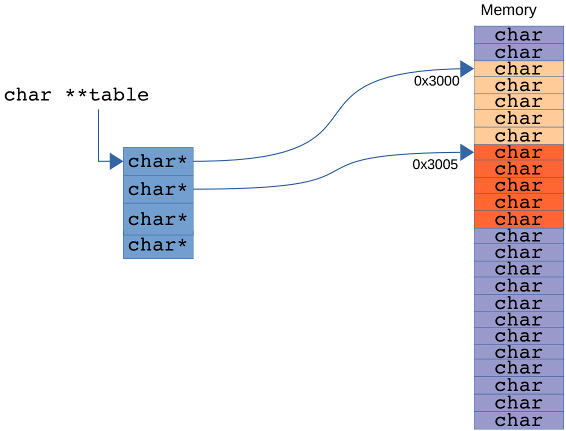
Allocating Space for a Multidimensional Array (1)



Case `i=0`

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

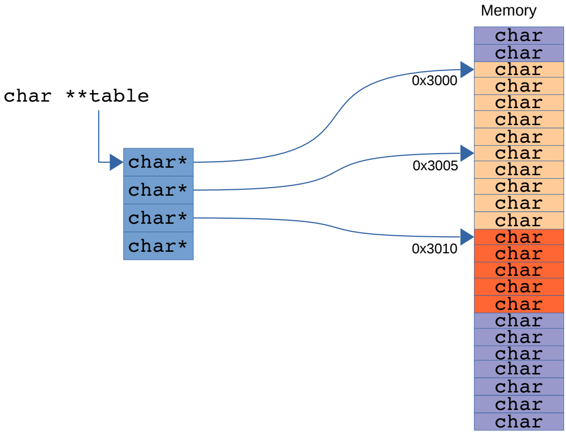
Allocating Space for a Multidimensional Array (2)



Case i=1

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

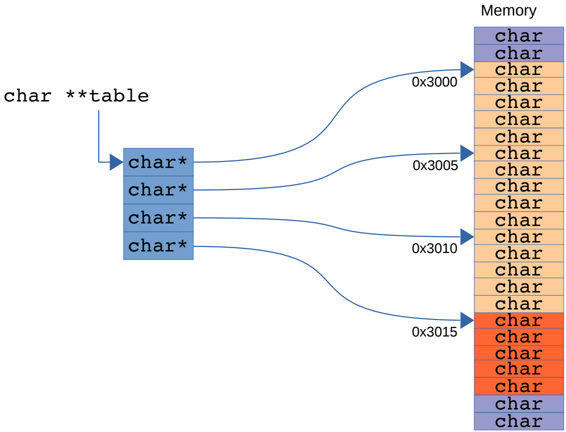
Allocating Space for a Multidimensional Array (3)



Case i=2

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

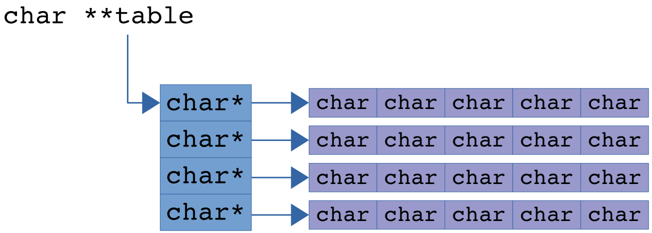
Allocating Space for a Multidimensional Array (4)



Case i=3

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

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




```c
for (i = 0; i < 4; i++)
  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

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

Working with 2D Dynamic Arrays

```
#include <stdio.h>
#include <stdlib.h>
void set_all_elements(int **arr, int numrow, int numcol) {
  int r, c;
  for (r = 0; r < numrow; r++)
    for (c = 0; c < numcol; c++)
      arr[r][c] = r * c;  // some value ...
}
int main() {
  int **table, row;
  table = (int **) malloc(sizeof(int *) * 3);
  if (table == NULL)
    exit(1);
  for (row = 0; row < 3; row++) {
    table[row] = (int *) malloc(sizeof(int) * 4);
    if (table[row] == NULL)
      exit(1);
    }
    set_all_elements(table, 3, 4);
}
```

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`