UESTC 1005 — Introductory Programming

Lecture 8 — Multidimensional Arrays and Pointers

Hasan T Abbas

Hasan.Abbas@glasgow.ac.uk

Mastering pointers is mastering the power of memory in C.

Lecture Outline

- Multidimensional Arrays
- Function Calls
- Pointers 🔑

Multidimensional Arrays

- 2D array akin to matrix (as in mathematics)

```
int table[5][5]; // creates a 2D array of 5 rows and 5 columns
```

- How much size does a 2D array take?

```
printf("size of array = %zu bytes\n", sizeof(table)); // %zu is for unsigned long
```

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	69	4	60	37	8	53
[1]	98	25	71	40	64	84
[2]	41	36	61	94	38	8
[3]	77	81	51	83	71	96
[4]	31	14	55	27	4	9
[5]	16	94	80	54	64	31

Use in Mathematics 📐

- 2D matrices heavily used in matrix manipulation
- Backbone of linear algebra and numerical computation

```c
#include <stdio.h>
#define N 5 // size of the matrix size is always 1 more
int main(void)
{

    int ident_matrix[N][N];
    int row, col;
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (row == col)
                ident_matrix[row][col] = 1;
            else
                ident_matrix[row][col] = 0;
    return 0;
}
```

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	1	0	0	0	0	0
[1]	0	1	0	0	0	0
[2]	0	0	1	0	0	0
[3]	0	0	0	1	0	0
[4]	0	0	0	0	1	0
[5]	0	0	0	0	0	1

Multidimensional Array Initialisation

```
int ident_matrix[5][5] = {{1, 1, 1, 0, 1},
            {0, 1, 0, 1, 1, 2},
            {0, 5, 7, 9, 10, 12},
            {3, 7, -1, 3, 32, 76},
            {0, 3, 5, -4, 1, 8}};
```

- Empty elements are initialised to `0`
- C treats multidimensional arrays as 1D essentially.
- We can go out of bounds in terms of indices.

```
printf("&array[%d][%d] = %p\n", row, col,  (void *)&ident_matrix[row][col]);
```

Function Calls and Arrays

- Recall the concept of passing by reference and by value

```
someFunction(ArrayName); // passing by reference. Array name is a pointer
```

Passing by Reference

- We pass the array name i.e., a pointer
- Since we pass the memory locations, contents are changed inside the function

Passing by value

- We pass individual elements of the array
- There is no pointer passed
- No changes made to the original array contents

```
someOtherFunction(AnotherArrayName[0]); // passing by value. Argument is an element
```

Pointers

Pointers 🔑

- Just another data type
- A key feature of C
- Allows byte-sized memory access
- A variable that stores the memory address of another variable.

```
int *ptr; // -> The variable ptr stores a pointer to an int
```

Pointer stores the address of a memory location

Why Pointers? 🤔

- Efficient memory management.
- Allows functions to modify variables directly.

So what is all about Pointers?

- How to create/declare pointers?
- `&` (address-of): Gets the memory address.
- `*` (dereference): Accesses the value at the memory address.

```
int a = 10;  // declare an integer variable
int *ptr; // declare a pointer
ptr = &a;  // and initialize it to the address of 'a'
printf("%d", *ptr);  // dereference the pointer and print the value it points to
```

Initialising and Indirect Assignment

- Pointers are like a map
- Variables are stored in memory, each at a unique address.
- Address Representation
- Pointers allow access to these addresses directly.
- E.g., `int x = 5;` & `int *p = &x;`
- `p` now holds the memory address of `x`.

Example 🔑

```c
#include <stdio.h> // which has definitions of printf function

int main() // void means nothing
{
    int a = 10;          // declare an integer variable
    int *ptr;            // declare a pointer
    ptr = &a;           // and initialize it to the address of 'a'
    printf("%d\n", *ptr); // dereference the pointer and print the value it points to
    printf("%p\n", ptr); // print the value the pointer points to
    printf("%p\n", &a); //  print the address of a
    printf("%p\n", a); //  print the value of a
    return 0;
}
```

Two Pointers

```c
#include <stdio.h> // which has definitions of printf function

int main() // void means nothing
{
    int a = 10; int b =20;
    int *ptr1, *ptr2;
    ptr1 = &a;
    ptr2 = &b;
    printf("%d\n", *ptr1);
    printf("%p\n", ptr2);
    printf("%p\n", &a);
    printf("%p\n", &b);
    return 0;
}
```

Pointer Arithmetic

- What happens when we perform arithmetic?
- We can simply *add*, *subtract* on pointers

```
char *p;
char a;
char b;
p = &a;
p += 1;
```

- Adds `1*sizeof(char)` to the memory address
- The pointer `p` now points to ?? 🤔

The Dangling Pointer ❗

- Null Pointer (`NULL`)
- A pointer that doesn't point to any address: `int *p = NULL;`
- A pointer pointing to deallocated memory.

Best Practice

- Always initialise pointers, and set to `NULL` after freeing memory.
- Avoid unexpected behaviour and crashes 🛑
- Memory area can be reused and data can be corrupted
- Modern compilers take care of this 😎
- start pointer names beginning with `p`, eg. `px`

Questions ❓

https://www.menti.com/alok51h26yh3

and type the code `5448 6697`.

Next Up ⏭️

- A wee Tutorial
- Pointers contd.
- Strings
- Bring your Laptops!