CSC209 Week 6 Notes

Hyungmo Gu

May 15, 2020

Struct 1 of 3

- Introducing Structs
  - **struct/structures** is like dictionary in Python or object in Javascript
  - there are differences between array and structure

	array	structure
data of same type	yes	not required
declaration details	type and number of elements (array [] notation)	types of members (struct keyword)
access via ...	index notation	dot notation

  - items in struct is called **member**
  - items in array is called **element**

```c
#include <stdio.h>
#include <string.h>

int main() {
    struct student {
        char first_name[20];
        char last_name[20];
        int year;
        float gpa;
    };

    struct student good_student;
```

```c

    strcpy(good_student.first_name, "Jo");
    strcpy(good_student.last_name, "Smith");
    good_student.year = 2;
    good_student.gpa = 3.2;

    printf("Name: %s %s\n", good_student.first_name, good_student.last_name);
    printf("Year %d. GPA %.2f\n", good_student.year, good_student.gpa);

    return 0;
}

```

Listing 1: struct_example_1.c

Struct 2 of 3

- Using Structs in Functions
  - Array pass function by **reference** (of the pointer of first element).
    - Changing value inside affects outside
  - Struct pass function by **value** like int and string.
    - Changing value in function doesn't affect value outside
    - Pointer used to pass by **reference**

```c
#include <stdio.h>
#include <string.h>

struct student {
    ...
};

void change(struct student *s) { // <- passes by reference
    ...
};

int main(void) {
    struct student good_student;
    ...
    change(&good_student); // <- to pass function by reference (This is too cool!!!)
    ...
    return 0;
}

```

Listing 2: struct_example_2.c

Struct 3 of 3

- Pointer to Structs
  - *(*p).student_name* is hard to define, and read
  - *p->student_name* is the same as above, but easier to read.
    - This is called **syntactic sugar**

```c
#include <stdio.h>
#include <string.h>

struct student {
    char first_name[20];
    char last_name[20];
    int year;
    float gpa;

};

int main(void) {
    struct student s;
    struct student *p;

    ...

    (*p).gpa = 3.0;
    p->year = 3; //<- HERE!!

    strcpy(p->first_name, "Hello");

    ...
    return 0;
}

```

Listing 3: struct_example_3.c

Dynamic memory allocation (malloc()) 1 of 5

- Introduction
  - Heap and Static Memory
    - **Heap memory:** Memory space controlled by programmer.
      - Programmer <u>must</u> clear memory after use
    - **Static memory:** Memory space controlled by computer

- Malloc
  * Allocates heap memory
  * Is in *stdlib* package
  * **Syntax:** void *malloc(size_t size);
    · returns pointer
    · *size_t* is int

```c
#include <stdio.h>
#include <stdlib.h>

int *set_i_heap() {
    int *j_pt = malloc(sizeof(int)); // <-HERE!!
    *j_pt = 5;
    return j_pt;
}

...

```

Listing 4: dynamic_mem_example_1.c

Dynamic memory allocation (malloc()) 2 of 5

- Allocating Memory on heap
  - **Syntax:** *heap_array = malloc(n * sizeof(type))

```c
#include <stdio.h>
#include <stdlib.h>

int *squares(int max_val) {
    int *result = malloc(max_val * sizeof(int)); // <- HERE :)

    for (int i = 0; i < max_val; i++){
        result[i] = (i+1)*(i+1);
    }

    return result;
}

...

```

Listing 5: dynamic_mem_example_2.c

Dynamic memory allocation (malloc()) 3 of 5

- Freeing Dynamically Allocated Memory
  - tells memory management the memory location is okay to be replaced.
  - doesn't remove memory from memory location
  - **Syntax:** void free(void *ptr)

```c
#include <stdio.h>
#include <stdlib.h>

int play_with_memory() {
    int i;
    int *pt = malloc(sizeof(int));

    i = 15;
    *pt = 15;

    free(pt); // <- Here :)

    return 0;
}
...

```

Listing 6: dynamic_mem_example_3.c

Dynamic memory allocation (malloc()) 4 of 5

- Returning an Address with a Pointer
  - **
    - Is called **double pointer**
    - First pointer used to store the address of variable
    - Second pointer used to store the address of the first pointer
    - Used to return something inside function to outside by reference
      - can be thought like using duc-taped 2 1m rulers to fetch something 2m away.

```c
#include <stdio.h>
#include <stdlib.h>

void helper(int **arr_double_p) { // <- 3. Double pointer. Another duc-taping.
    *arr_double_p = malloc(sizeof(int) * 3); // 4. <- the other end of 2 1m rulers
```

```c

    int *arr = *arr_double_p;

    arr[0] = 0;
    arr[1] = 21;
    arr[2] = 23;
}

int main() {
    int *data; // <- 1. Firs of 2 1m rulers
    helper(&data); // <- 2. Note how memory address of pointer is passed. Think of this as ductaping

    printf("middle value: %d\n", data[1]);

    free(data);
    return 0;
}

```

Listing 7: dynamic_mem_example_4.c

Dynamic memory allocation (malloc()) 5 of 5

- Nested Data Structures
  - Use Case: Arrays in Array
    - *free* needs to be used on all elements in heap memory.
    - 7 Stars Tip: write free as writing nested data structures
      - Or the future Moe will restart the computer, come back, and say Bad Moe, Bad!!

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int **double_pointers = malloc(sizeof(int) * 2);

    double_pointers[0] = malloc(sizeof(int));
    double_pointers[1] = malloc(sizeof(int) * 2);

    double_pointers[0][0] = 12;
    double_pointers[1][0] = 2;
    double_pointers[1][1] = 3;
    double_pointers[1][2] = 4;

    free(double_pointers[0]);
    free(double_pointers[1]);
```

```
        free(double_pointers);
    }

```

Listing 8: dynamic_mem_example_5.c