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**

```
1   #include <stdio.h>
2   #include <string.h>
3
4   int main() {
5       struct student {
6           char first_name[20];
7           char last_name[20];
8           int year;
9           float gpa;
10      };
11
12      struct student good_student;
```

```
13
14          strcpy(good_student.first_name, "Jo");
15          strcpy(good_student.last_name, "Smith");
16          good_student.year = 2;
17          good_student.gpa = 3.2;
18
19          printf("Name: %s %s\n", good_student.first_name, good_student.
    last_name);
20          printf("Year %d. GPA %.2f\n", good_student.year, good_student.
    gpa);
21
22          return 0;
23     }
24
```

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**
        ```
        1      #include <stdio.h>
        2      #include <string.h>
        3
        4      struct student {
        5          ...
        6      };
        7
        8      void change(struct student *s) { // <- passes by reference
        9          ...
        10     };
        11
        12     int main(void) {
        13         struct student good_student;
        14         ...
        15         change(&good_student); // <- to pass function by
            reference (This is too cool!!!)
        16         ...
        17         return 0;
        18     }
        19
        ```
        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

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    int *set_i_heap() {
5        int *j_pt = malloc(sizeof(int)); // <-HERE!!
6        *j_pt = 5;
7        return j_pt;
8    }
9
10   ...
11
12
```

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))

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    int *squares(int max_val) {
5        int *result = malloc(max_val * sizeof(int)); // <- HERE :)
6
7        for (int i = 0; i < max_val; i++){
8            result[i] = (i+1)*(i+1);
9        }
10
11       return result;
12   }
13
14   ...
15
```

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)

```
1     #include <stdio.h>
2     #include <stdlib.h>
3
4     int play_with_memory() {
5         int i;
6         int *pt = malloc(sizeof(int));
7
8         i = 15;
9         *pt = 15;
10
11        free(pt); // <- Here :)
12
13        return 0;
14    }
15    ...
16
```

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.

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

```c
 6
 7        int *arr = *arr_double_p;
 8
 9        arr[0] = 0;
10        arr[1] = 21;
11        arr[2] = 23;
12    }
13
14    int main() {
15        int *data; // <- 1. Firs of 2 1m rulers
16        helper(&data); // <- 2. Note how memory address of pointer
       is passed. Think of this as ductaping
17
18        printf("middle value: %d\n", data[1]);
19
20        free(data);
21        return 0;
22    }
23
```

Listing 7: dynamic_mem_example_4.c