# L6 - Structures

December 14, 2019

## 1 Structures

A *structure* consists of one or more variables, grouped together *under a single name*, so we can deal with them as a single unit. The variables in a structure may have different types. For example, for a point in 3-dimensional cartesian space; we can:

```
In [ ]: struct point{
            int x;
            int y;
            int z;
        };
```

In this case, `point` is **not** a variable - it is a *structure tag*, or a type. x, y and z are *members* of the structure `point`. The struct declaration does not allocate memory and creates no actual variables or storage of state. To actually make some points, we can:

```
In [ ]: struct point point1, point2;
```

And just like normal variables, these structs can be initialized with constant expressions as part of a variable declaration:

```
In [ ]: struct point estick = { 320, 200, 100 }; // sure, that'll work
```

And we can initialize a a struct after declaration, if we *cast* the expression to the type:

```
In [ ]: struct 2dpoint noodle;

        point4 = { 100, 200, 50 } // not like this

        point4 = (struct point) {100, 200, 50 }; // like this
```

**Type Casting** The structure in the above uses the *casting* operator to change the type of `point4`.

We can also use `typedef` to make a synonymous declaration for a struct:

```
In [ ]: struct 3dpoint{
            int x;
            int y;
            int z;
        };

        typedef struct 3dpoint pointy;
```

wherein we have `pointy` now as an alias or synonym for `3dpoint`. This also **does not** generate a variable with type `pointy` or `struct point`. We can now use `pointy` anywhere we might have used `3dpoint`.

Structures, when passed into functions, are *passed by value*. Structure members (x, y, z above) can be accessed individuially. Structures can be copied and assigned. Structures can be returned by functions.

```
In [ ]: point4.x // the x-value of point 4

        point4.x = point4.x + 20; // add twenty to point4's x value

        point3.y += 30; // add thirty to point3's y value

        point2[z]; // this won't work
                   // cannot subscript a structure like an array
```

```
In [ ]: point1.x = 200;
        point1.y = 100;
        point1.z = 50;

        point2 = point1; // point2 is a new point with the same location as point1
```

The following code uses a struct which contains both an integer student number and an array of marks - recall that members need not be of the same type.

```
In [7]: #include <stdio.h>
        #include <stdlib.h>

        const int SIZE = 5;   // Maximum array size

        typedef struct
        {
            int id;
            float mark [5];
        } student_t;

        void change_marks(float marks[], int SIZE)
        {
            for (int i = 0; i < SIZE; i++)
            {
                marks[i] = marks[i] + 10;
            }
        }


        int main (void)
        {
            student_t student1;          // A structure consists of an array
            int i;                         // loop index
```

```
        // Initialize the array in the structure
        student1.id = 100500800;
        for (i = 0; i < SIZE; i++) {
            student1.mark [i] = i + 70;
        }

        // Print the array of structures - points
        printf ("student ID: %d\n", student1.id);
        for (i = 0; i < SIZE; i++) {
            printf("mark [%d] = %.2f\n", i, student1.mark[i]);
        }

        change_marks(student1.mark, 5);

        printf("\n======================\n\n"); // separator line

        // Print the array of structures again after changing
        printf ("student ID: %d\n", student1.id);
        for (i = 0; i < SIZE; i++) {
            printf("mark [%d] = %.2f\n", i, student1.mark[i]);
        }

        return 0;
    }
```

```
student ID: 100500800
mark [0] = 70.00
mark [1] = 71.00
mark [2] = 72.00
mark [3] = 73.00
mark [4] = 74.00

======================

student ID: 100500800
mark [0] = 80.00
mark [1] = 81.00
mark [2] = 82.00
mark [3] = 83.00
mark [4] = 84.00
```

But what if we changed the function such that it took the whole `student` struct, rather than just the `mark` array?

```
In [8]: #include <stdio.h>
        #include <stdlib.h>
```

```c
const int SIZE = 5;   // Maximum array size

typedef struct
{
    int id;
    float mark [5];
} student_t;

void change_marks(student_t student, int SIZE)
{
    for (int i = 0; i < SIZE; i++)
    {
        student.mark[i] = student.mark[i] + 10;
    }
}


int main (void)
{
    student_t student1;          // A structure conssits of an array
    int i;                         // loop index

    // Initialize the array in the structure
    student1.id = 100500800;
    for (i = 0; i < SIZE; i++) {
        student1.mark [i] = i + 70;
    }

    // Print the array of structures - points
    printf ("student ID: %d\n", student1.id);
    for (i = 0; i < SIZE; i++) {
        printf("mark [%d] = %.2f\n", i, student1.mark[i]);
    }

    change_marks(student1, 5);

    printf("\n=====================\n\n"); // separator line

    // Print the array of structures again after changing
    printf ("student ID: %d\n", student1.id);
    for (i = 0; i < SIZE; i++) {
        printf("mark [%d] = %.2f\n", i, student1.mark[i]);
    }

    return 0;
}
```

```
student ID: 100500800
mark [0] = 70.00
mark [1] = 71.00
mark [2] = 72.00
mark [3] = 73.00
mark [4] = 74.00

======================

student ID: 100500800
mark [0] = 70.00
mark [1] = 71.00
mark [2] = 72.00
mark [3] = 73.00
mark [4] = 74.00
```

Because the marks array was passed into the function *by reference*, it changed the array in place. The struct is passed into the function *by value*, so the array is changed only in function scope and the global array remains **unchanged!**

### 1.1 Structures and Functions

```
In [ ]: // declaration
        3dpoint make3dpoint( int x, int y, int z)
        {
            3dpoint temp;

            temp.x = x;
            temp.y = y;
            temp.z = z;

            return temp;
        }

        // more concisely
        3dpoint make3dpoint( int x, int y, int z)
        {
            return (point_t) { x, y };
        }

        // typical call
        int a, b;

        3dpoint pointy;

        a = 300;
        b = 200;
```

```
pointy = makepoint(a, b);
```

In [ ]:
```
#include <stdio.h>
#include <stdlib.h>

struct 2dpoint
{
    int x;
    int y;
};

2dpont makepoint(int x, int y)
{
    return (2dpoint) { x, y }; // cast to 2dpoint using the casting operator
}

2dpoint addpoints(2dpoint pt1, 2dpoint pt2)
{
    pt1.x = pt1.x + pt2.x; // pt1 won't be modified because it is passed by value!
    pt1.y = pt1.y + pt2.y;
}

int main (void)
{

}
```