

# C language

Imad Kissami<sup>1</sup>

<sup>1</sup>Mohammed VI Polytechnic University, Benguerir, Morocco



# Structures

## Introduction

- A structure can be used to define a new data type that combines different types into a single (compound) data type
  - Definition is similar to a template or blueprint
  - Composed of members of previously defined types
- Structures must be defined before use
- C has three different methods to define a structure
  - variable structures
  - tagged structures
  - type-defined structures

# Structures

## Struct variable

- A variable structure definition defines a struct variable

```
1 struct {  
2     double x; // x coordinate  
3     double y; // y coordinate  
4 } point;
```

- *point* is the variable name
- *x* and *y* are the structure members
- **DON'T FORGET THE SEMICOLON**

# Structures

## Tagged Structure

- A tagged structure definition defines a type
- We can use the tag to define variables, parameters, and return types

```
1 struct point_t{  
2     double x; // x coordinate  
3     double y; // y coordinate  
4 };
```

- *point\_t* is the Structure tag
- *x* and *y* are the structure members
- **DON'T FORGET THE SEMICOLON**

- Variable definitions:

```
1 struct point_t point1, point2, point3;
```

- Variables *point1*, *point2*, and *point3* all have members *x* and *y*.

# Structures

## Typedef Structure

- A typed-defined structure allows the definition of variables without the struct keyword.
- We can use the tag to define variables, parameters, and return types.

```
1 typedef struct{
2     long ssn; // Social Security Number
3     int empType; // Employee Type
4     float salary; // Annual Salary
5 } employee_t;
```

- *employee\_t* is the **New type name**
- *ssn*, *empType* and *salary* are the **structure members**
- **DON'T FORGET THE SEMICOLON**

- Variable definition:

```
1 employee_t emp;
```

- Variable *emp* has members *ssn*, *empType*, and *salary*.

# Structures

## Dot Operator (.)

- Used to access member variables

- Syntax:

```
1 structure_variable_name.member_name
```

- These variables may be used like any other variables

```
1 struct point_t{  
2     double x; // x coordinate  
3     double y; // y coordinate  
4 };
```

- Example of usage:

```
1 void setPoints(){  
2     struct point_t point1, point2;  
3     point1.x = 7; // Init point1 members  
4     point2.y = 11;  
5  
6     point2 = point1; // copy point1 to point2  
7     ...  
8 }
```

# Structures

## Arrow Operator (→)

- Used to access member variables using a pointer

- Arrow Operator Syntax:

```
1 structure_variable_pointer->member_name
```

- Dot Operator Syntax:

```
1 (*structure_variable_pointer).member_name
```

```
1 typedef struct{
2     long ssn; // Social Security Number
3     int empType; // Employee Type
4     float salary; // Annual Salary
5 } employee_t;
```

- Example of usage:

```
1 employee_t *newEmp(long n, int type, float sal){
2     employee_t * empPtr = malloc(sizeof(employee_t));
3
4     empPtr->ssn = n; // -> operator
5     empPtr->empType = type; // -> operator
6     (*empPtr).salary = sal; // dot operator
7
8     return empPtr;
9 }
```

# Structures

## Nested Structures

- A member that is of a structure type is nested

```
1 typedef struct {
2     int month;
3     int day;
4     int year;
5 } date_t;
6
7 typedef struct {
8     double height;
9     int weight;
10    date_t birthday;
11 } personInfo_t;
12
13 // Define variable of type personInfo_t
14 personInfo_t person;
15 ...
16
17 // person.birthday is a member of person
18 // person.birthday.year is a member of person.birthday
19
20 printf("Birth year is %d\n", person.birthday.year);
```



# Structures

## Initializing Structures

- A structure may be initialized at the time it is declared
- Order is essential
  - The sequence of values is used to initialize the successive variables in the struct
- It is an error to have more initializers than members
- If fewer initializers than members, the initializers provided are used to initialize the data members
- The remainder are initialized to 0 for primitive types

```
1 typedef struct {  
2     int month;  
3     int day;  
4     int year;  
5 } date_t;  
6  
7 date_t due_date = {31, 03, 2022};
```

# Structures

## Dynamic Allocation of Structures

- The `sizeof()` operator should always be used in dynamic allocation of storage for structured data types and in reading and writing structured data types

```
1 #include <stdio.h>
2 #include <stdlib.h> // for calloc
3 int main(){
4     typedef struct {
5         int month;
6         int day;
7         int year;
8     } date_t;
9
10    date_t due_date;
11
12    int date_t_len = sizeof(date_t); // sizeof type
13    int date_du_len = sizeof(due_date); // sizeof variable
14
15    date_t * due_dates = calloc(100, sizeof(date_t));
16
17    printf("sizeof(date_t)=%d\n", date_t_len);
18    printf("sizeof(date_du)=%d\n", date_du_len);
19
20    return 0;
21 }
```

```
1 sizeof(date_t)=12
2 sizeof(date_du)=12
```

# Structures

## Arrays Within Structures

- A member of a structure may be an array

```
1 typedef struct {  
2     long ssn; // SSN  
3     double payRate; // Hourly rate  
4     float hoursWorked[7]; // Daily hours worked Sunday-Saturday  
5 } timeCard_t;  
6  
7 timeCard_t empTime;  
8  
9 empTime.hoursWorked[5] = 6.5; Thursday hours worked
```

# Structures

## Arrays of Structures

- We can also create an array of structure types

```
1 typedef struct {  
2     // unsigned char will hold 0-255  
3     unsigned char red;  
4     unsigned char green;  
5     unsigned char blue;  
6 } pixel_t;  
7  
8 pixel_t pixelMap[800][600];  
9  
10 pixelMap[425][37].red = 127;  
11 pixelMap[425][37].green = 0;  
12 pixelMap[425][37].blue = 58;
```

# Structures

## Arrays of Structures Containing Arrays

- We can also create an array of structures that contain arrays

```
1 typedef struct {
2     long ssn; // SSN
3     double payRate; // Hourly rate
4     float hoursWorked[7]; // Daily hours worked Sun-Sat
5 } timeCard_t;
6
7 timeCard_t empTime[100];
8
9 // Thur hour worked, emp # 10
10
11 empTime[9].hoursWorked[5] = 6.5;
```

# Structures

## Structures as Parameters

- A struct, like an int, may be passed to a function
- The process works just like passing an int, in that:
  - The complete structure is copied to the stack
  - Called function is unable to modify the caller's copy of the variable

```
1 typedef struct {
2     double x; // x coordinate
3     double y; // y coordinate
4 } point_t;
5
6 void changePoint(point_t p){
7     printf("x=%.1lf, y=%.1lf\n", p.x, p.y);
8     p.x = 3.4;
9     p.y = 4.5;
10 }
11
12 void main(){
13
14     point_t point = {1.2, 2.3};
15     changePoint(point);
16     printf("x=%.1lf, y=%.1lf\n", point.x, point.y);
17
18 }
```

```
1 x=1.2, y=2.3
2 x=1.2, y=2.3
```

# Structures

## Structures as Parameters

- Disadvantage of passing structures by value: Copying large structures onto stack
  - Is inefficient
  - May cause stack overflow

```
1 typedef struct {  
2     int w[1000*1000*1000]; // one billion int elements  
3 } big_t;  
4  
5 // Passing a variable of type big_t will cause  
6 // 4 billion bytes to be copied to the stack  
7  
8 big_t fourGB;  
9  
10 int i;  
11  
12 for (i=0; i < 1000000; i++) // 1,000,000 times  
13     slow_call(fourGB)
```

# Structures

## Structures as Parameters

- More efficient: Pass the address of the struct
- Passing an address requires that only a single word be pushed on the stack, no matter the size
  - Called function can then modify the structure.

```
1 typedef struct {
2     double x; // x coordinate
3     double y; // y coordinate
4 } point_t;
5
6 void changePoint(point_t * p){
7     printf("x=%.1lf, y=%.1lf\n", p->x, p->y);
8     p->x = 3.4;
9     p->y = 4.5;
10 }
11
12 void main(){
13
14     point_t point = {1.2, 2.3};
15     changePoint(&point);
16     printf("x=%.1lf, y=%.1lf\n", point.x, point.y);
17
18 }
```

```
1 x=1.2, y=2.3
2 x=3.4, y=4.5
```



# Structures

## Const Struct Parameter

- What if you do not want the recipient to be able to modify the structure?
  - Use the const modifier

```
1 typedef struct {
2     double x; // x coordinate
3     double y; // y coordinate
4 } point_t;
5
6 void changePoint(const point_t * p){
7     printf("x=%.1lf, y=%.1lf\n", p->x, p->y);
8     p->x = 3.4;
9     p->y = 4.5;
10 }
11
12 void main(){
13     point_t point = {1.2, 2.3};
14     changePoint(point);
15     printf("x=%.1lf, y=%.1lf\n", point->x, point->y);
16 }
```

```
1 example4.c: In function 'changePoint':
2 example4.c:10:8: error: assignment of member 'x' in read-only object
3 10 | p->x = 3.4;
4    | ^
5 example4.c:11:8: error: assignment of member 'y' in read-only object
6 11 | p->y = 4.5;
7    | ^
```

# Structures

## Return Structure

- Scalar values (*int, float, etc*) are efficiently returned in CPU registers
- Historically, the structure assignments and the *return* of structures was not supported in C
- But, the return of pointers (*addresses*), including pointers to structures, has always been supported

# Structures

## Return Structure

### ■ Example:

```
1 typedef struct {
2     // unsigned char will hold 0-255
3     unsigned char red;
4     unsigned char green;
5     unsigned char blue;
6 } pixel_t;
7
8 pixel_t * getEmptyPixel(){
9     // empty pixel = zeros
10    pixel_t p = {0, 0, 0};
11    // return pointer to empty pixel
12    return &p;
13 }
14
15 void main(){
16     pixel_t ePixel, *pixelPtr;
17     pixelPtr = getEmptyPixel();
18     // Immediately use return
19     ePixel = *pixelPtr;
20 }
```

```
1 example5.c: In function 'getEmptyPixel':
2 example5.c:14:10: warning: function returns address of local variable [-Wreturn-local-addr]
3 14 | return &p;
4    | ~~~~~
```

# Structures

## Return Structure Pointer to Local Variable

- Reason: function is returning a pointer to a variable that was allocated on the stack during execution of the function
- Such variables are subject to being wiped out by subsequent function calls

# Structures

## Function Return Structure Values

- It is possible for a function to return a structure.
- This facility depends upon the structure assignment mechanisms which copies one complete structure to another.
  - Avoids the unsafe condition associated with returning a pointer, but
  - Incurs the possibly extreme penalty of copying a very large structure

```
1 typedef struct {
2     // unsigned char will hold 0-255
3     unsigned char red;
4     unsigned char green;
5     unsigned char blue;
6 } pixel_t;
7
8 pixel_t getEmptyPixel(){
9     // empty pixel = zeros
10    pixel_t p = {0, 0, 0};
11    // return pointer to empty pixel
12    return p;
13 }
14
15 void main(){
16     pixel_t ePixel;
17     ePixel = getEmptyPixel();
18 }
```

# Structures

## Arrays as Parameters & Return

- Array's address is passed as parameter
  - Simulates passing by reference
- Embedding array in structure
  - The only way to pass an array by value is to embed it in a structure
  - The only way to return an array is to embed it in a structure
  - Both involve copying
    - \* Beware of size