

Structures in C

By Abdallah Ghazy

- في لغة C ، ال structure هو نوع بيانات مخصص بنعزفه بنفسنا. بنستخدم الكلمة المفتاحية struct عشان نعزف نوع بيانات مخصص يجمع مجموعة من العناصر ذات أنواع مختلفة.
- الفرق بين ال array وال structure هو إن ال array هو مجموعة متجانسة من عناصر متشابهة في النوع، يعني كل العناصر في ال array من نفس النوع، لكن في ال structure، ممكن يكون فيه عناصر بأنواع مختلفة متخزنة بجانب بعضها، وكل عنصر منها له اسم يميزه (يعني غير متجانسة).
- في حالات كثير بنحتاج نتعامل مع قيم من أنواع بيانات مختلفة لها علاقة ببعضها.
- ال struct بيسمحك تجمع القيم دي كلها في متغير واحد وتتعامل معها بطريقة منظمة وسهلة.

```
// تعريف هيكل للكتاب
typedef struct {
    char title[100];    // عنوان الكتاب
    char author[100];   // مؤلف الكتاب
    double price;        // سعر الكتاب
    int pages;           // عدد صفحات الكتاب
} Book;
```

مثلاً، الكتاب بنوصفه بعنوانه (string)، مؤلفه (string)، سعره (double)، وعدد صفحاته (integer)، وهكذا. بدلاً من استخدام أربع متغيرات مختلفة، ممكن نخزن القيم دي كلها في متغير واحد من نوع struct

Syntax of Structure Declaration

```
struct [structure tag]{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

- The structure tag is optional and each member definition is a normal variable definition, such as "int i;" or "float f;" or any other valid variable definition.
- At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

Example

```
struct book{
    char title[50];
    char author[50];
    double price;
    int pages;
} book1;
```

Structure Variable Declaration

علشان توصل لل members في ال structure وتتعامل معاها، لازم تعرّف متغير من نوع ال structure الأول. لتعريف متغير من نوع ال structure، اكتب اسم ال structure مع الكلمة المفتاحية "struct" متبوعاً باسم متغير ال structure. المتغير ده هيسخدم للوصول والتعامل مع ال members.

Example

The following statement demonstrates how to declare (create) a structure variable

```
struct book book1;
```

Structure Initialization

The initialization of a struct variable is done by placing the value of each element inside curly brackets.

Example

```
struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};
```

Accessing the Structure Members

To access the members of a structure, first, you need to declare a structure variable and then use the **dot (.) operator** along with the structure variable.

Example 1

```
#include <stdio.h>

struct book{
    char title[10];
    char author[20];
    double price;
    int pages;
};

int main(){
    struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};

    printf("Title:  %s \n", book1.title);
    printf("Author:  %s \n", book1.author);
    printf("Price:   %lf\n", book1.price);
    printf("Pages:   %d \n", book1.pages);
    printf("Size of book struct: %d", sizeof(struct book));
    return 0;
}
```

```
Title: Learn C
Author: Dennis Ritchie
Price: 675.500000
Pages: 325
Size of book struct: 48
```

Copying Structures

The **assignment (=) operator** can be used to copy a structure directly. You can also use the assignment operator (=) to assign the value of the member of one structure to another.

Example

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

struct book{
    char title[10];
    char author[20];
    double price;
    int pages;
};

int main(){
    struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325}, book2;
    book2 = book1;

    printf("Title: %s \n", book2.title);
    printf("Author: %s \n", book2.author);
    printf("Price: %lf \n", book1.price);
    printf("Pages: %d \n", book1.pages);
    printf("Size of book struct: %d", sizeof(struct book));
    return 0;
}
```

Title: Learn C
Author: Dennis Ritchie
Price: 675.500000
Pages: 325
Size of book struct: 48

Find the size of a structure.

```
# include <stdio.h>

struct example {
    int a;
    char b;
    char str_name[20];
    double c;
    float d;
};

int main() {
    struct example e;
    printf("Size of example structure: %ld bytes\n", sizeof(e));
    printf("\nSize occupied by int a: %d\n", sizeof(e.a));
    printf("Size occupied by char b: %d\n", sizeof(e.b));
    printf("Size occupied by string str_name: %d\n", sizeof(e.str_name));
    printf("Size occupied by double c: %d\n", sizeof(e.c));
    printf("Size occupied by float d: %d\n", sizeof(e.d));
    return 0;
}
```

<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy
Size of example structure: 48 bytes

Size occupied by int a: 4
Size occupied by char b: 1
Size occupied by string str_name: 20
Size occupied by double c: 8
Size occupied by float d: 4

The size of a structure is not always equal to the sum of the sizes of its members due to the following reasons:

- **Padding for Alignment:** Compilers may add padding between members to ensure proper alignment in memory.
- **Alignment Requirements:** These requirements vary depending on CPU architecture and compiler options.
- **Effect of Padding:** The amount of padding added can increase the overall size of the structure.

Structures and Functions in C

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

Passing the Structure by Value

When you pass a structure by value, a copy of the structure is made inside the function. Any changes made inside the function do not affect the original structure.

Example

```
#include <stdio.h>

// Define the structure
struct Book {
    char title[100];
    char author[100];
    double price;
    int pages;
};

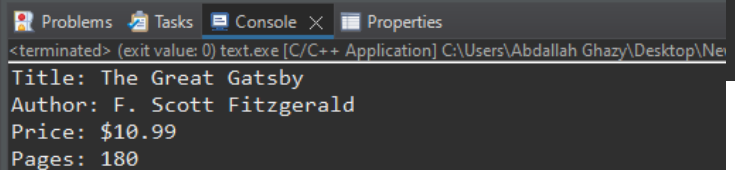
// Function to print book details
void printBook(struct Book book) {
    printf("Title: %s\n", book.title);
    printf("Author: %s\n", book.author);
    printf("Price: $%.2f\n", book.price);
    printf("Pages: %d\n", book.pages);
}

int main() {
    // Declare a structure variable
    struct Book myBook;

    // Initialize the book details
    snprintf(myBook.title, sizeof(myBook.title), "The Great Gatsby");
    snprintf(myBook.author, sizeof(myBook.author), "F. Scott Fitzgerald");
    myBook.price = 10.99;
    myBook.pages = 180;

    // Call the function and pass the structure as an argument
    printBook(myBook);

    return 0;
}
```



The screenshot shows a console window with the following output:

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\Nei
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Price: $10.99
Pages: 180
```

Passing the Structure by Reference

When you pass a structure by reference, you pass the address of the structure to the function. Any changes made inside the function directly affect the original structure.

Example

```
#include <stdio.h>

// Define the structure
struct Book {
    char title[100];
    char author[100];
    double price;
    int pages;
};

// Function to update book details
void updateBook(struct Book *book) {
    // Update the book details
    snprintf(book->title, sizeof(book->title), "1984");
    snprintf(book->author, sizeof(book->author), "George Orwell");
    book->price = 9.99;
    book->pages = 328;
}

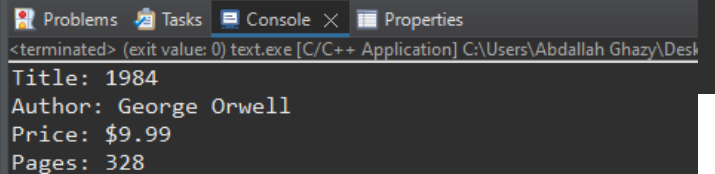
int main() {
    // Declare a structure variable
    struct Book myBook;

    // Initialize the book details
    snprintf(myBook.title, sizeof(myBook.title), "The Great Gatsby");
    snprintf(myBook.author, sizeof(myBook.author), "F. Scott Fitzgerald");
    myBook.price = 10.99;
    myBook.pages = 180;

    // Call the function and pass the address of the structure as an argument
    updateBook(&myBook);

    // Print the updated details
    printf("Title: %s\n", myBook.title);
    printf("Author: %s\n", myBook.author);
    printf("Price: $%.2f\n", myBook.price);
    printf("Pages: %d\n", myBook.pages);

    return 0;
}
```



The screenshot shows a console window with the following output:

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop
Title: 1984
Author: George Orwell
Price: $9.99
Pages: 328
```

Pass Struct Elements

A derived type is a combination of one or more elements of any of the primary types as well as another derived type. It is possible to pass elements to a function, either by value or by reference.

Example

```
#include <stdio.h>

struct rectangle{
    float len, brd;
};

int area(float, float);

int main(){

    struct rectangle r;
    r.len = 10.50; r.brd = 20.5;
    area(r.len, r.brd);

    return 0;
}

int area(float a, float b){

    double area = (double)(a*b);
    printf("Length: %f \nBreadth: %f \nArea: %lf\n", a, b, area);

    return 0;
}
```

Length: 10.500000
Breadth: 20.500000
Area: 215.250000

Return Struct from a Function

```
#include <stdio.h>

struct rectangle {
    float len, brd;
    double area;
};

struct rectangle area(float x, float y);

int main(){

    struct rectangle r;
    float x, y;

    x = 10.5; y = 20.5;
    r = area(x, y);

    printf("Length: %f \n Breadth: %f \n Area: %lf\n", r.len, r.brd, r.area);

    return 0;
}

struct rectangle area(float x, float y){

    double area = (double)(x*y);
    struct rectangle r = {x, y, area};

    return r;
}
```

Problems Tasks Console × Properties
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Gha:
Length: 10.500000
Breadth: 20.500000
Area: 215.250000

Array of Structures in C

Initializing a Struct Array

```
struct book{
    char title[10];
    double price;
    int pages;
};
```

you can declare an array and initialize it by giving the values of each element inside curly brackets. Each element in the struct array is a struct value itself.

```
struct book b[3] = {
    {"Learn C", 650.50, 325},
    {"C Pointers", 175, 225},
    {"C Pearls", 250, 250}
};
```

How does the compiler allocate memory for this array?

Since we have an array of three elements, of struct whose size is 32 bytes, the array occupies "32 x 3" bytes. Each block of 32 bytes will accommodate a "title", "price" and "pages" element.

L	E	A	R	N		C				675.50	325
C	P	O	I	N	T	E	R	S		175	225
C		P	E	A	R	L	S			250	250

Declaring a Struct Array

```
struct book b[3];
strcpy(b[0].title, "Learn C");
b[0].price = 650.50;
b[0].pages=325;

strcpy(b[1].title, "C Pointers");
b[1].price = 175;
b[1].pages=225;

strcpy(b[2].title, "C Pearls");
b[2].price = 250;250
b[2].pages=325;
```

Dot (.) Operator in C

the dot (.) operator in C language is also known as "**direction selection member**".

It is used to **select** members of structure and union.

The dot (.) operator is a **binary operator** that requires two operands (**structure or union name** and **member name**) and it has the highest operator precedence.

```
var.member;
```

Example

```
struct newtype {  
    type elem1;  
    type elem2;  
    type elem3;  
    ...  
    ...  
};
```

You can then declare a variable of this derived data type as –

```
struct newtype var;
```

To access a certain member,

```
var.elem1;
```


Nested Structures in C

A structure within a structure is known as **nested structure in C**. When one of the elements in the definition of a struct, type is of another struct type, then we call it a nested structure in C. Nested structures are defined when one of the elements of a struct type is itself a composite representation of one or more types.

Nested Structure Declaration

Syntax

A general syntax of a nested structure is as follows –

```
struct struct1{
    type var1;
    type var2;
    struct struct2 strvar;
}
```

Example

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

struct dob{
    int d, m, y;
};

struct employee{
    char name[10];
    float salary;
    struct dob d1;
};

int main(){

    struct employee e1 = {"Kiran", 25000, {12, 5, 1990}};
    printf("Name: %s\n", e1.name);
    printf("Salary: %f\n", e1.salary);
    printf("Date of Birth: %d-%d-%d\n", e1.d1.d, e1.d1.m, e1.d1.y);

    return 0;
}
```

```
Name: Kiran
Salary: 25000.000000
Date of Birth: 12-5-1990
```

Dot Operator with Nested Structure

Nested structures are defined when one of the elements of a struct type is itself a composite representation of one or more types.

The dot operator can also be used to access the members of nested structures (and union types also). It can be done in the same way as done for the normal structure.

```
struct struct1 {  
    var1;  
    var2;  
    struct struct2 {  
        var3;  
        var4;  
    } s2;  
} s1;
```

```
s1.s2.var3;
```

Example

```
#include <stdio.h>  
  
struct employee {  
    char name[10];  
    float salary;  
    struct dob {  
        int d, m, y;  
    } d1;  
};  
  
int main(){  
  
    struct employee e1 = {"Kiran", 25000, {12, 5, 1990}};  
  
    printf("Name: %s\n", e1.name);  
    printf("Salary: %f\n", e1.salary);  
    printf("Date of Birth: %d-%d-%d\n", e1.d1.d, e1.d1.m, e1.d1.y);  
  
    return 0;  
}
```

```
Name: Kiran  
Salary: 25000.000000  
Date of Birth: 12-5-1990
```

Accessing the Members Using the Arrow Operator

C also has another method to access the members of a struct variable. It can be done with the arrow operator (->) with the help of a pointer to the struct variable.

```
struct newtype {  
    type elem1;  
    type elem2;  
    type elem3;  
    ...  
    ...  
};
```

You can then declare a variable of this derived data type, and its pointer as –

```
struct newtype var;  
struct newtype *ptr=&var;
```

To access a certain member through the pointer, use the syntax

```
ptr->elem1;
```

Example

```
#include <stdio.h>  
  
struct book {  
    char title[10];  
    double price;  
    int pages;  
};  
  
int main (){  
  
    struct book b1 = {"Learn C", 675.50, 325};  
    struct book *strptr;  
    strptr = &b1;  
  
    printf("Title: %s\n", strptr->title);  
    printf("Price: %lf\n", strptr->price);  
    printf("No of Pages: %d\n", strptr->pages);  
  
    return 0;  
}
```

```
Title: Learn C  
Price: 675.500000  
No of Pages: 325
```

- The dot operator (.) is used to access the struct elements via the struct variable.
- To access the elements via its pointer, we must use the indirection operator (->).

Bit Fields in C

When we declare a struct or a union type, the size of the struct/union type variable depends on the individual size of its elements.

Instead of the default memory size, you can set the size the bits to restrict the size. The specified size is called **bit fields**.

Bit Field Declaration

```
struct {  
    type [member_name] : width ;  
};
```

Element	Description
type	An integer type that determines how a bit-field's value is interpreted. The type may be int , signed int , or unsigned int .
member_name	The name of the bit-field.
width	Number of bits in the bit-field. "width" must be less than or equal to bit width of specified type.

Example

```
#include <stdio.h>  
  
/* define simple structure */  
struct {  
    unsigned int widthValidated;  
    unsigned int heightValidated;  
} status1;  
  
/* define a structure with bit fields */  
struct {  
    unsigned int widthValidated : 1;  
    unsigned int heightValidated : 1;  
} status2;  
  
int main() {  
  
    printf("Memory size occupied by status1: %d\n", sizeof(status1));  
    printf("Memory size occupied by status2: %d\n", sizeof(status2));  
  
    return 0;  
}
```

Memory size occupied by status1: 8
Memory size occupied by status2: 4

Example

```
#include <stdio.h>

struct {
    unsigned int age : 3;
} Age;

int main() {

    Age.age = 4;
    printf("Sizeof(Age): %d\n", sizeof(Age));
    printf("Age.age: %d\n", Age.age);

    Age.age = 7;
    printf("Age.age : %d\n", Age.age);

    Age.age = 8;
    printf("Age.age : %d\n", Age.age);

    return 0;
}
```

Output

When the above code is compiled, it will compile with a warning –

```
warning: unsigned conversion from 'int' to 'unsigned char:3' changes value from '8' to '0' [-Woverflow]
```

And, when executed, it will produce the following output –

```
Sizeof(Age): 4
Age.age: 4
Age.age: 7
Age.age: 0
```

C has a built-in feature called bit-fields to access a single bit.

❖ Uses of Bit-fields:

- Store several Boolean (true/false) variables in one byte when storage is limited.
- Encode status information from certain devices into one or more bits within a byte.
- Access bits within a byte for certain encryption routines.

❖ Bit-field Requirements:

- Must be a member of a structure or union.
- Defines how long, in bits, the field is.

❖ Bit-field Definition:

- General form: type name: length;
- type must be int, signed, unsigned, or _Bool (C99).

- ❖ Frequently used when analyzing input from a hardware device.

Example

```
#include <stdio.h>

struct status_type {
    unsigned char delta_cts:1;
    unsigned char delta_dsr:1;
    unsigned char tr_edge:1;
    unsigned char delta_rec:1;
    unsigned char cts:1;
    unsigned char dsr:1;
    unsigned char ring:1;
    unsigned char rec_line:1;
} status;

int main(int argc, char **argv) {
    status.cts = 1;
    printf("sizeof structure = %d", sizeof(status));
    return 0;
}
```

Problems Tasks Console X Properties
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy
sizeof structure = 1

VS

```
#include <stdio.h>

struct status_type {
    unsigned int delta_cts:1;
    unsigned int delta_dsr:1;
    unsigned int tr_edge:1;
    unsigned int delta_rec:1;
    unsigned int cts:1;
    unsigned int dsr:1;
    unsigned int ring:1;
    unsigned int rec_line:1;
} status;

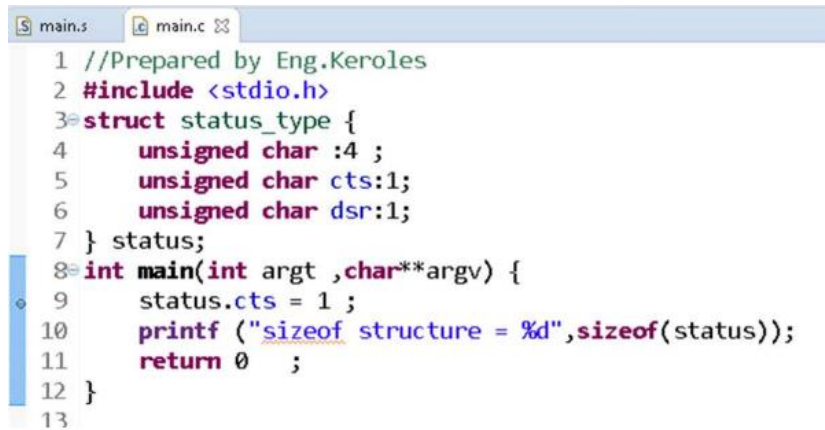
int main(int argc, char **argv) {
    status.cts = 1;
    printf("sizeof structure = %d", sizeof(status));
    return 0;
}
```

Problems Tasks Console X Properties
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\De
sizeof structure = 4

You do not have to name each bit-field.

This makes it easy to reach the bit you want, bypassing unused ones.

- For example, if you only care about the cts and dsr bits, you could declare the status_type structure like this:
- Also, notice that the bits after dsr do not need to be specified if they are not used.



```
1 //Prepared by Eng.Keroles
2 #include <stdio.h>
3 struct status_type {
4     unsigned char :4 ;
5     unsigned char cts:1;
6     unsigned char dsr:1;
7 } status;
8 int main(int argc ,char**argv) {
9     status.cts = 1 ;
10    printf ("sizeof structure = %d",sizeof(status));
11    return 0 ;
12 }
13
```

Your points about bit-fields are accurate. Here's a clearer summary of those restrictions:

- 1) **Addressing Bit-Fields:** You cannot take the address of a bit-field. This means you can't use the & operator to get a pointer to a bit-field, which limits certain types of pointer operations.
- 2) **Arraying Bit-Fields:** Bit-fields cannot be used in arrays. Each bit-field must be part of a struct, and they cannot be indexed or iterated over like traditional arrays.
- 3) **Bit-Field Order:** The order of bit-fields (whether they are packed from right to left or left to right) can vary between different machines or compilers. This lack of consistency can lead to portability issues.
- 4) **Machine Dependencies:** Bit-fields are subject to machine-specific and compiler-specific behavior, which means their implementation details can differ across platforms. This can affect how bit-fields are packed, aligned, and accessed.

These limitations are important to consider when designing systems that rely on bit-fields for low-level data manipulation.



"من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"