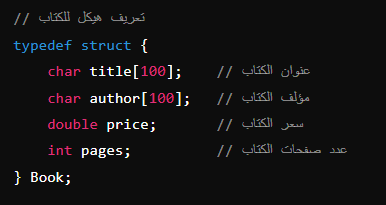
**Structures in C**

**By Abdallah Ghazy**

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



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**

****

* 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**

Structure Variable Declaration

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

# Example

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

Structure Initialization

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

# Example

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

# 

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

# 

Find the size of a structure.

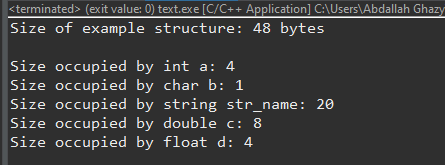
**# include** <stdio.h>

**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;

}

## **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");

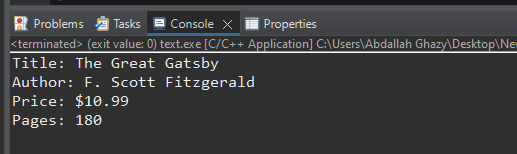
**snprintf**(myBook.author, **sizeof**(myBook.author), "F. Scott Fitzgerald");

myBook.price = 10.99;

myBook.pages = 180;

// Call the function and pass the structure as an argument

**printBook**(myBook);



**return** 0;

}

#### 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);

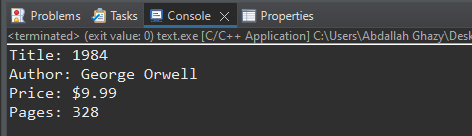
// Print the updated details

**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;

}

#### 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

# 

# 

 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);

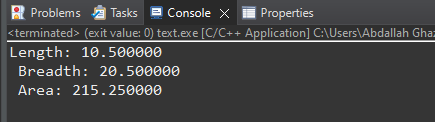
**printf**("Length: %f \n Breadth: %f \n Area: %lf\n", r.len, r.brd, r.area);

**return** 0;

}

**struct** **rectangle** **area**(**float** x, **float** y){

**double** area = (**double**)(x\*y);

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

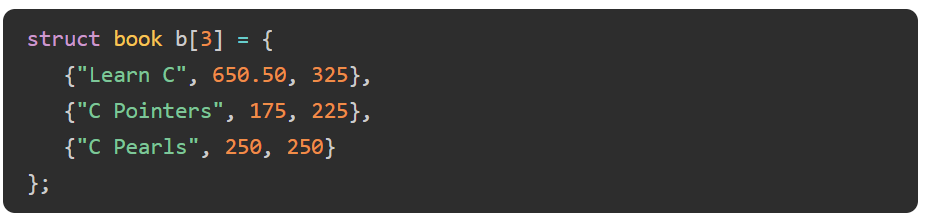
**return** r;

}

Array of Structures in C

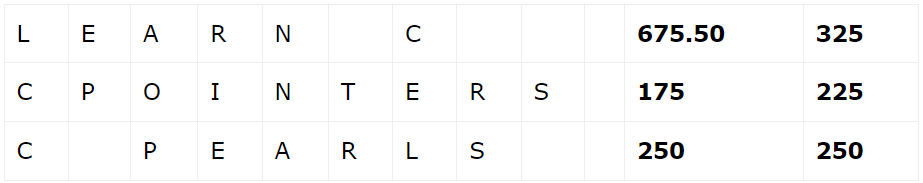
Initializing a Struct Array

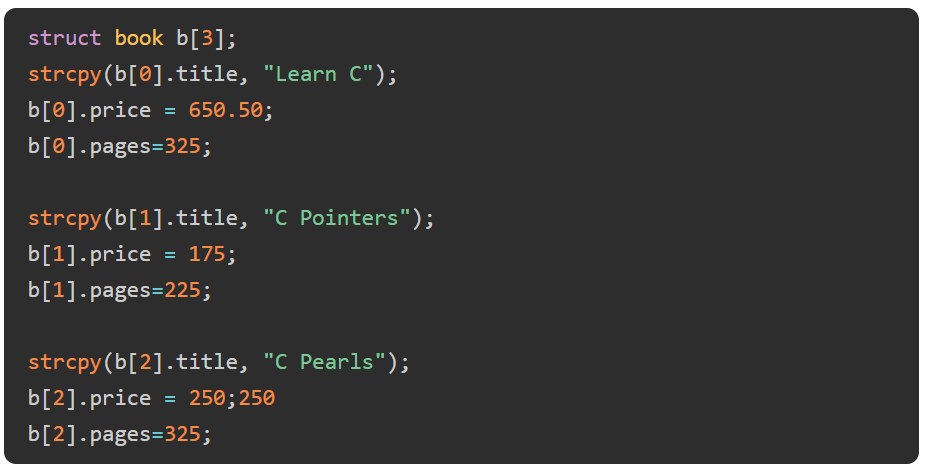
## you can declare an array and initialize it by giving the values of each element inside curly brackets. Each element in the struct array is a struct value itself.



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.



Declaring a Struct Array

Dot (.) Operator in C

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

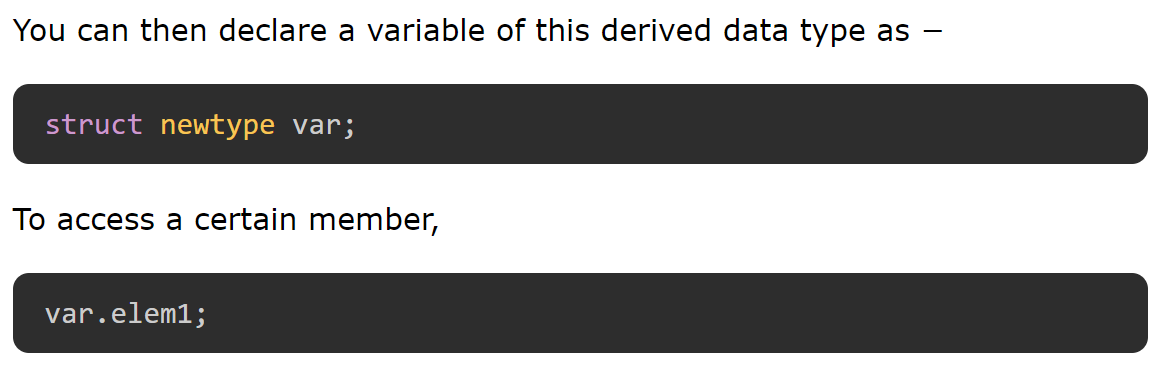
## It is used to **select** members of structure and union.

## 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.



# Example

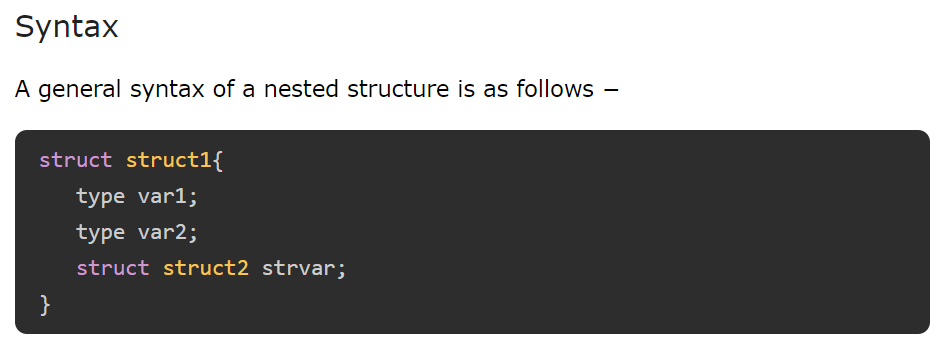
# 



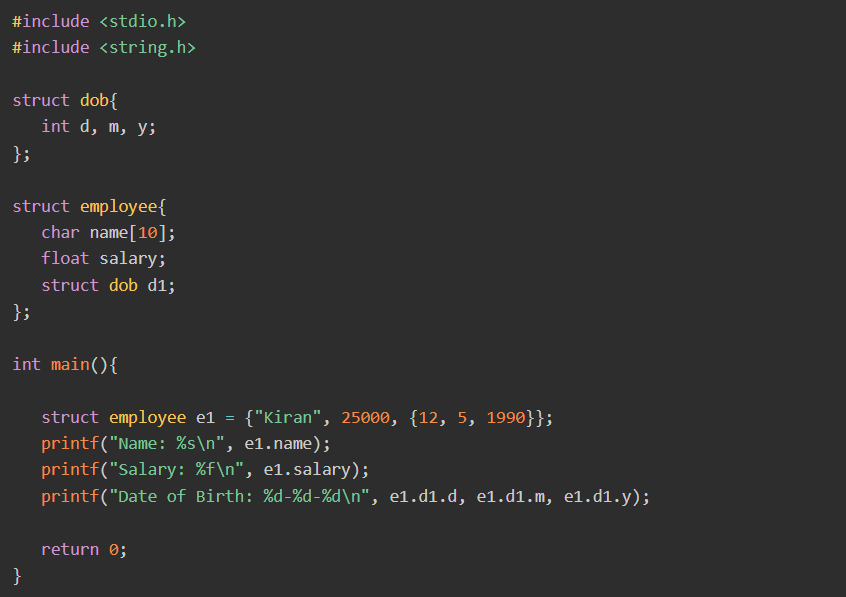
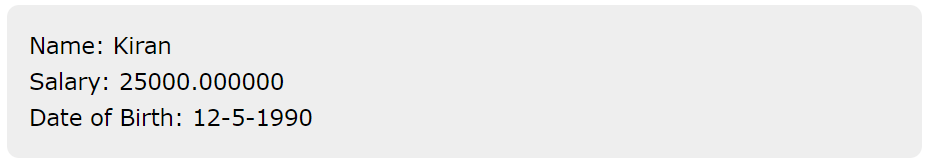
Nested Structures in C

## 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



# Example



Dot Operator with Nested Structure

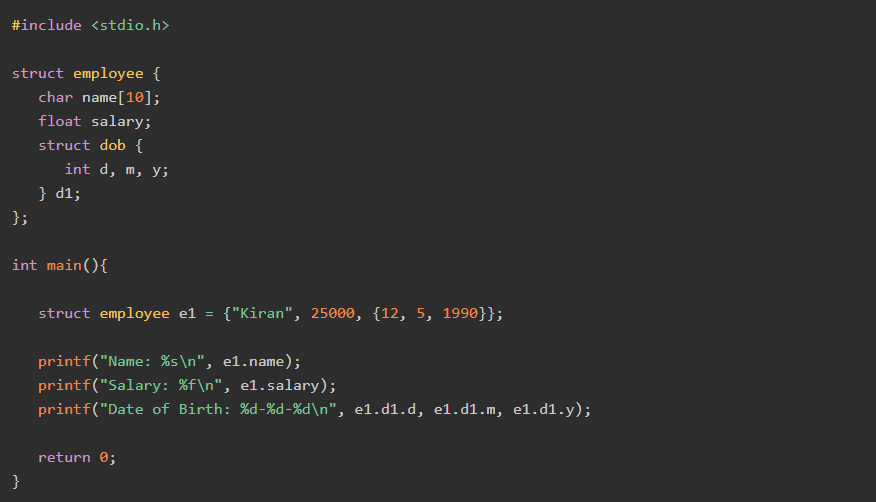
## **Nested structures** are defined when one of the elements of a struct type is itself a composite representation of one or more types.

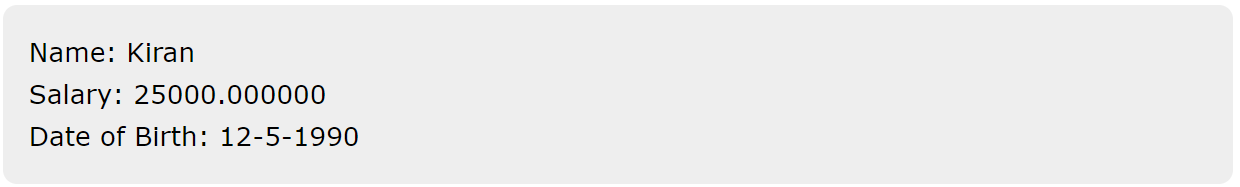
## The dot operator can also be used to access the members of nested structures (and union types also). It can be done in the same way as done for the normal structure.





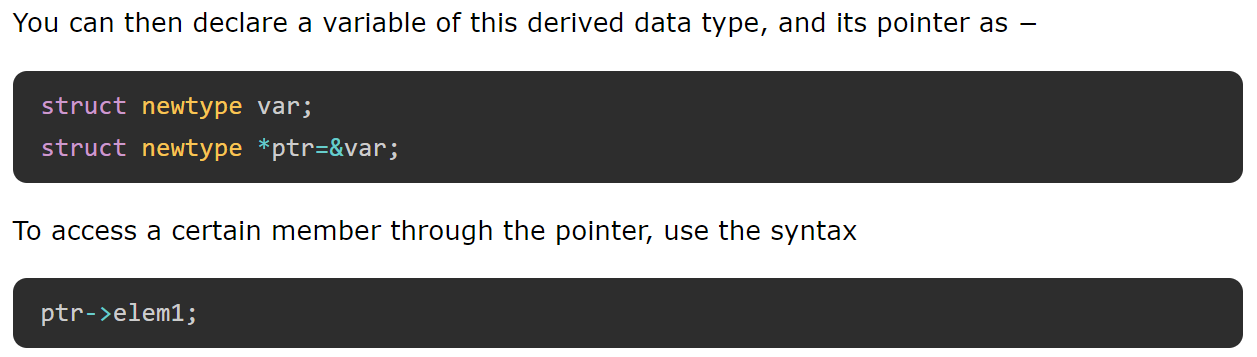
# Example



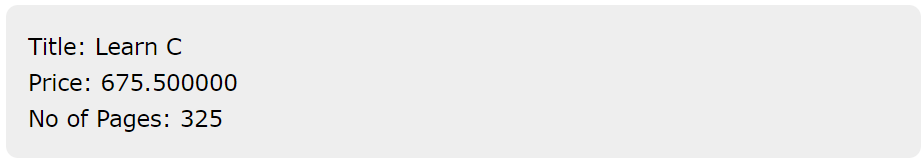


Accessing the Members Using the Arrow Operator

## 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.



# Example



## **The dot operator (.)** is used to access the struct elements via the struct variable.

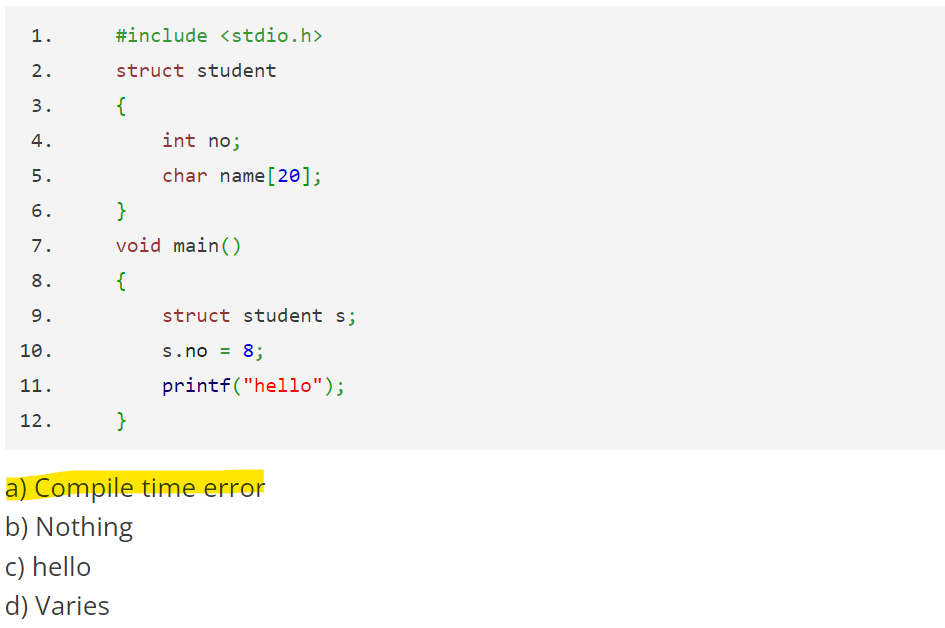
## To access the elements via its pointer, we must use the indirection **operator (->).**

# Questions and Answers

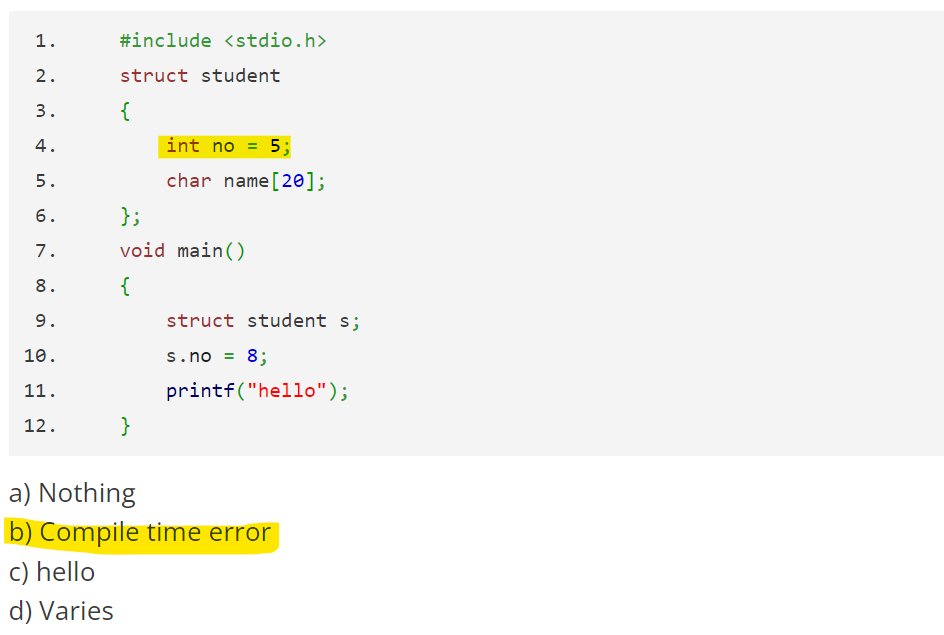
# Which of the following structure declaration will throw an error?

# 

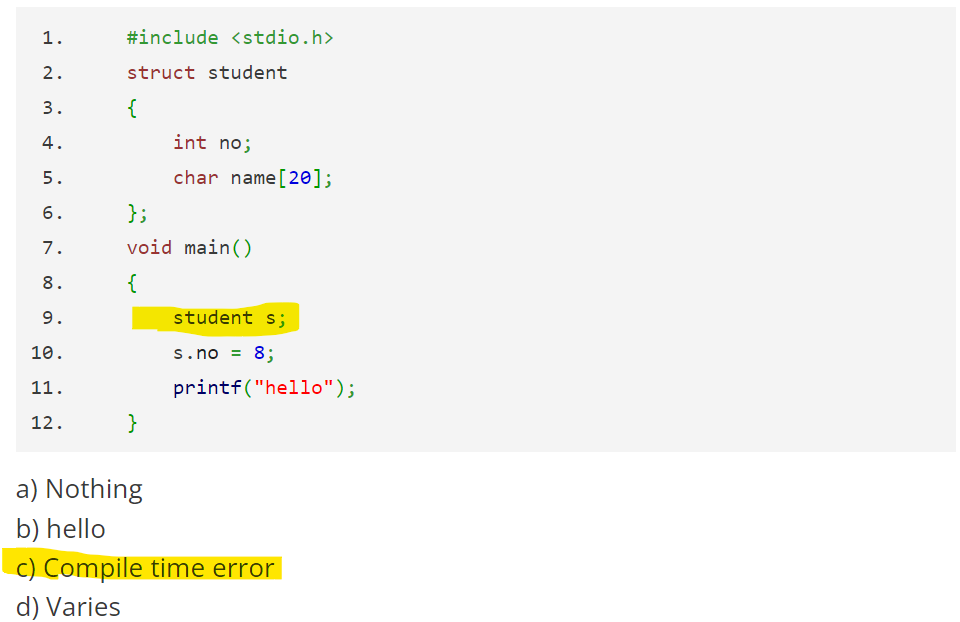
What will be the output of the following C code?



#### What will be the output of the following C code?



 What will be the output of the following C code?



#### Can the following C code be compiled successfully?

#### 

#### 

#### How many bytes in memory taken by the following C structure?

#### 

## The int k is placed at offset 0 (occupying 4 bytes).

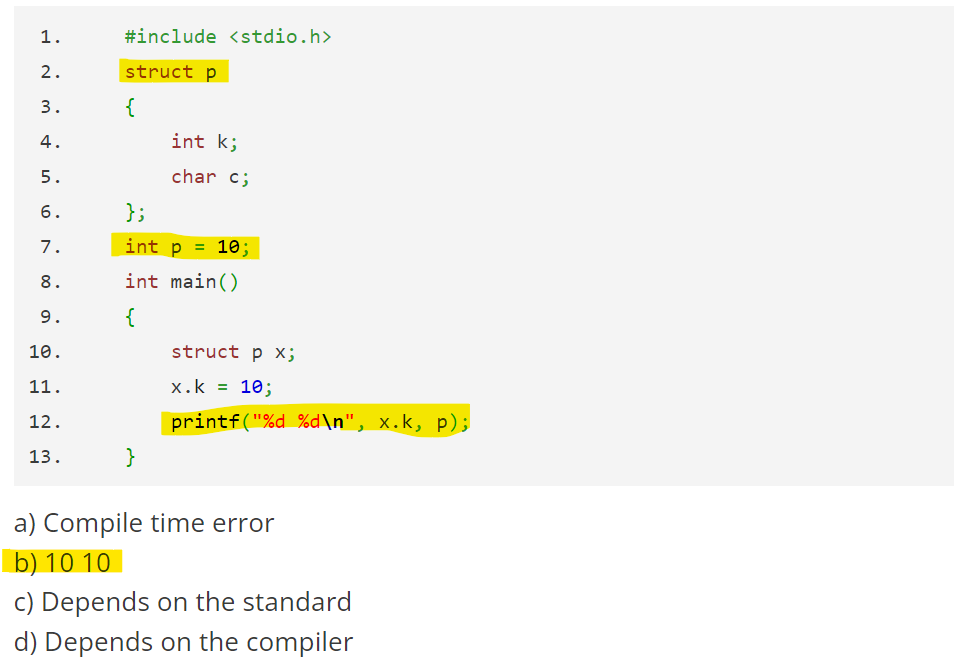
## The char c is placed at offset 4 (occupying 1 byte).

## To ensure the next structure in an array starts at a multiple of the largest member's size (4 bytes), the compiler will add 3 bytes of padding after char c.

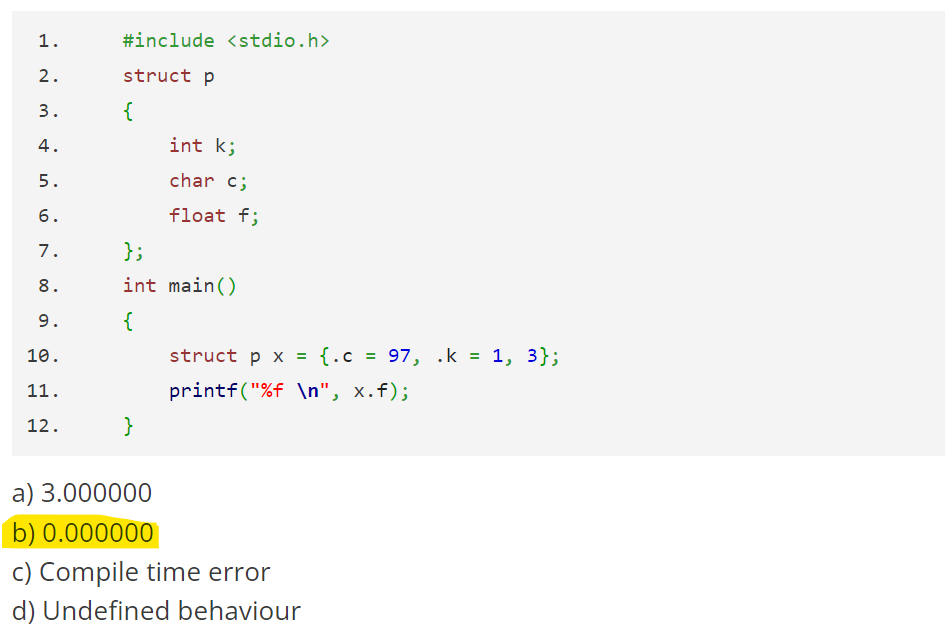
## 4 bytes (int k) + 1 byte (char c) + 3 bytes (padding) = 8 bytes.

What will be the output of the following C code?

#### 

What will be the output of the following C code?

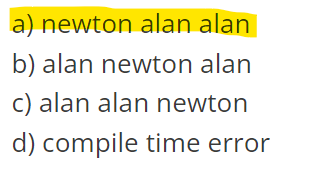
 What will be the output of the following C code according to C99 standard?



# Questions and Answers – Structures and Functions

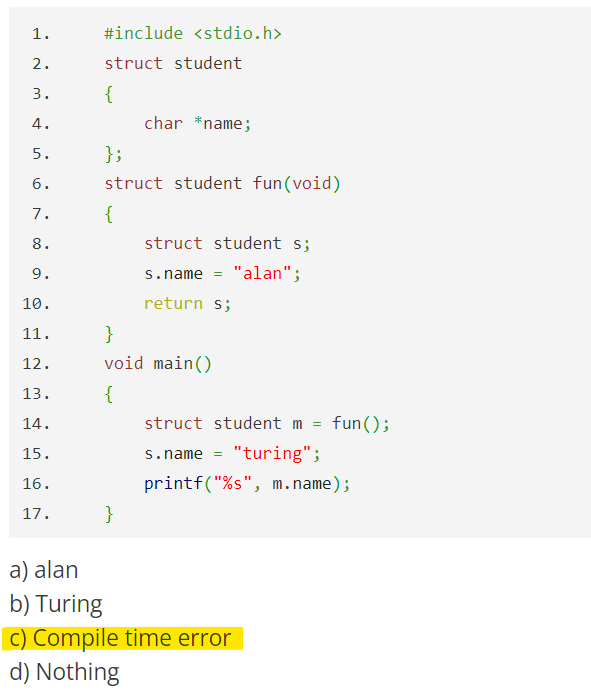
# What will be the output of the following C code?

# 

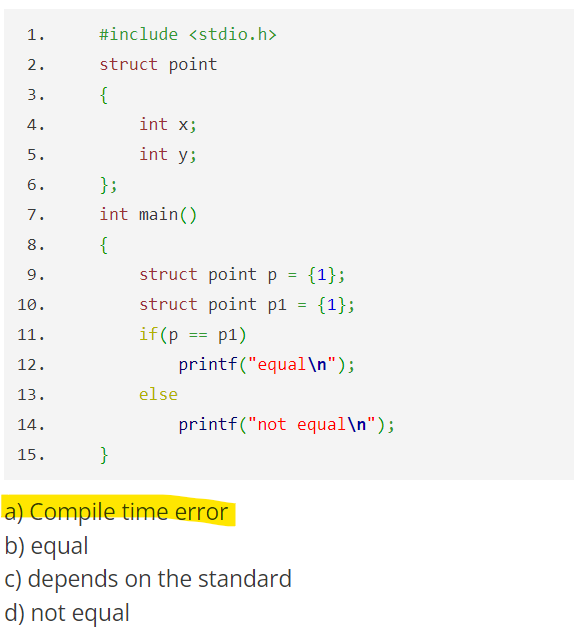


What will be the output of the following C code?

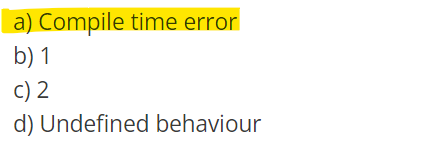
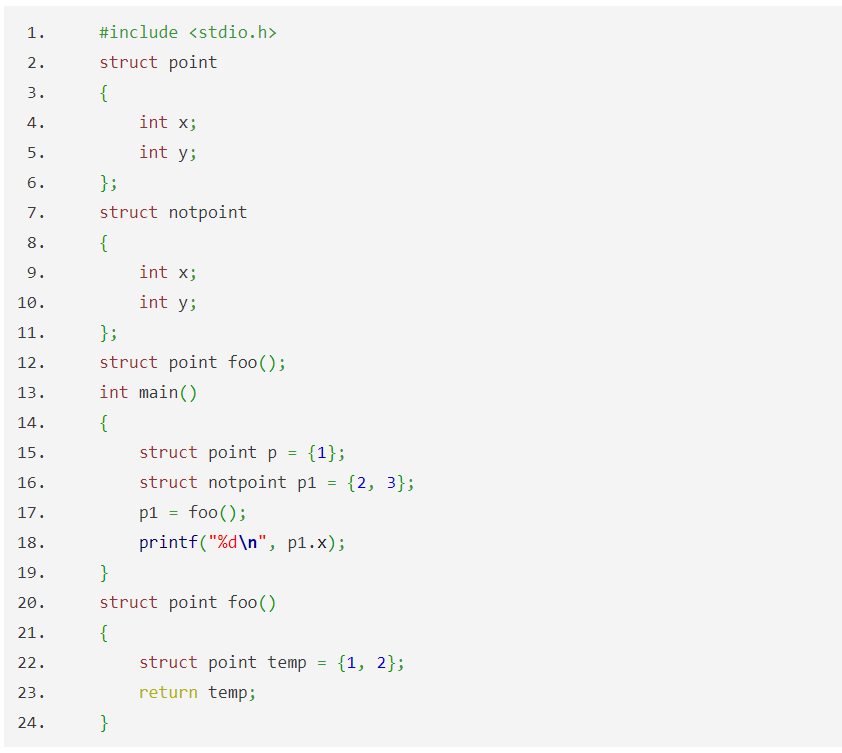


What will be the output of the following C code?

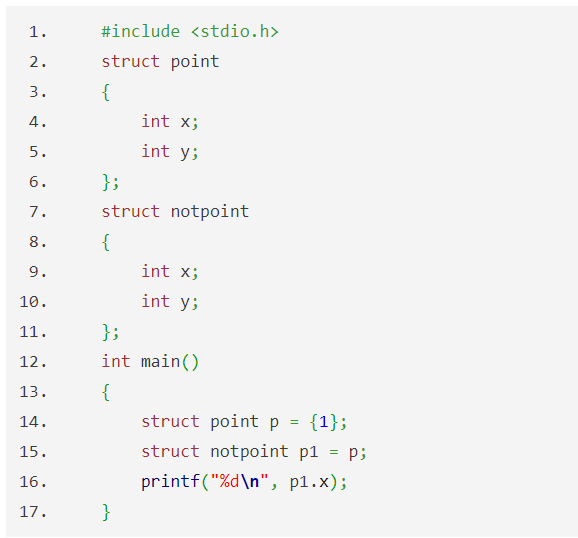
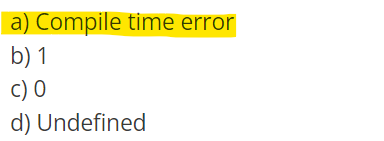
What will be the output of the following C code?



What will be the output of the following C code?



What will be the output of the following C code?



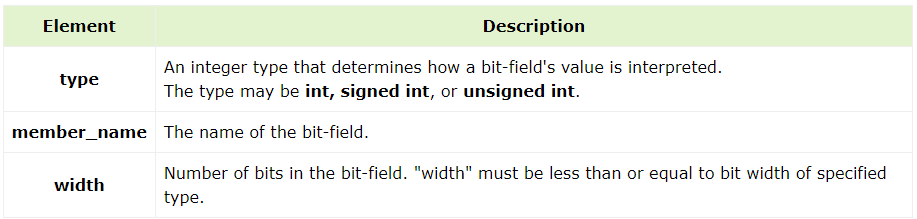
 What will be the output of the following C code?

#### Bit Fields in C

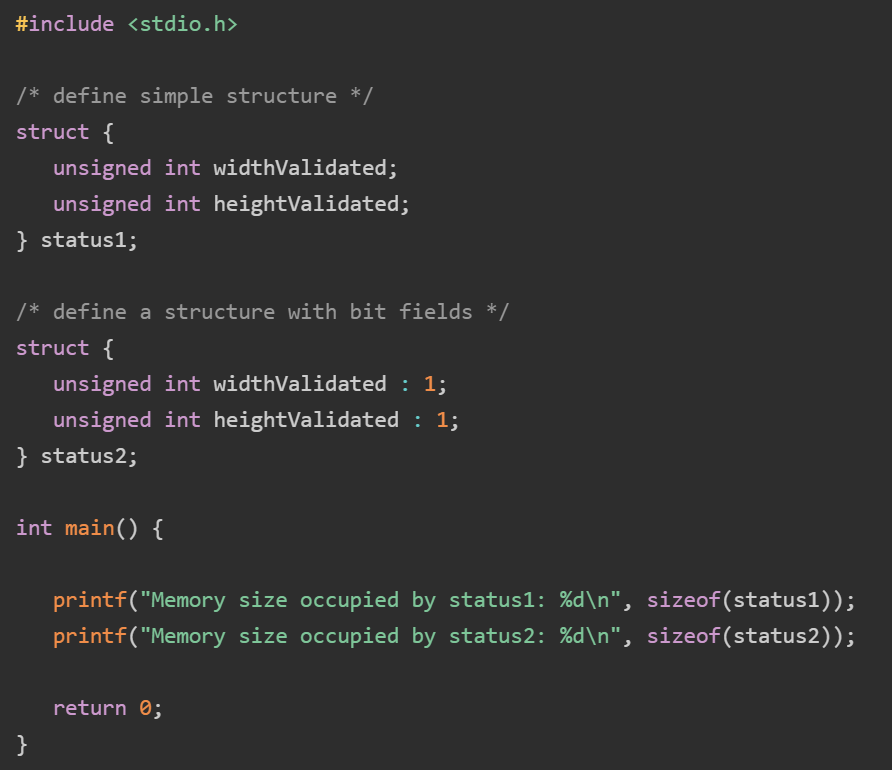
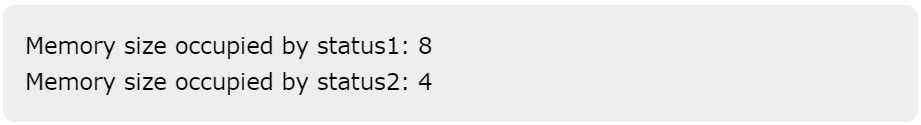
## When we declare a struct or a union type, the size of the struct/union type variable depends on the individual size of its elements.

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

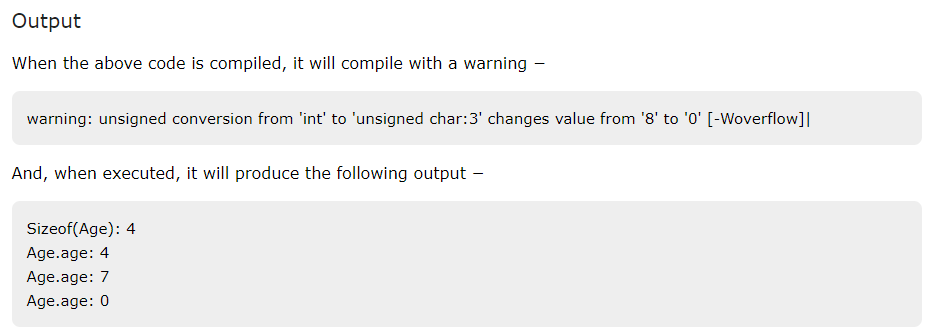
Bit Field Declaration



# Example



# Example



## **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;

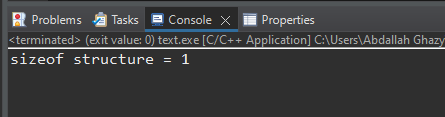
**unsigned** **char** rec\_line:1;

} status;

**int** **main**(**int** argc, **char** \*\*argv) {

status.cts = 1;

**printf**("sizeof structure = %d", **sizeof**(status));

 **return** 0;

}

**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;

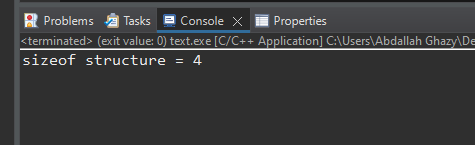
**unsigned** **int** rec\_line:1;

} status;

**int** **main**(**int** argc, **char** \*\*argv) {

status.cts = 1;

**printf**("sizeof structure = %d", **sizeof**(status));

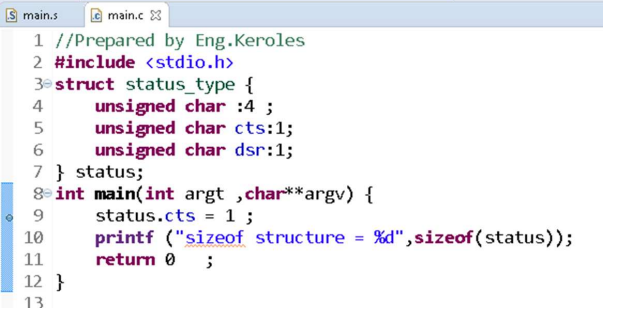
 **return** 0;

}

**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.



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

## **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.

## **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.

## **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.

## **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.

[](https://github.com/Abdallah-Ghazy)

[](https://www.linkedin.com/in/abdallah-ghazy/)

[](https://www.facebook.com/profile.php?id=100009485341470)

[](https://www.youtube.com/channel/UCRh59pwh7KTLgfftifu_zrQ)

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