Strctures:

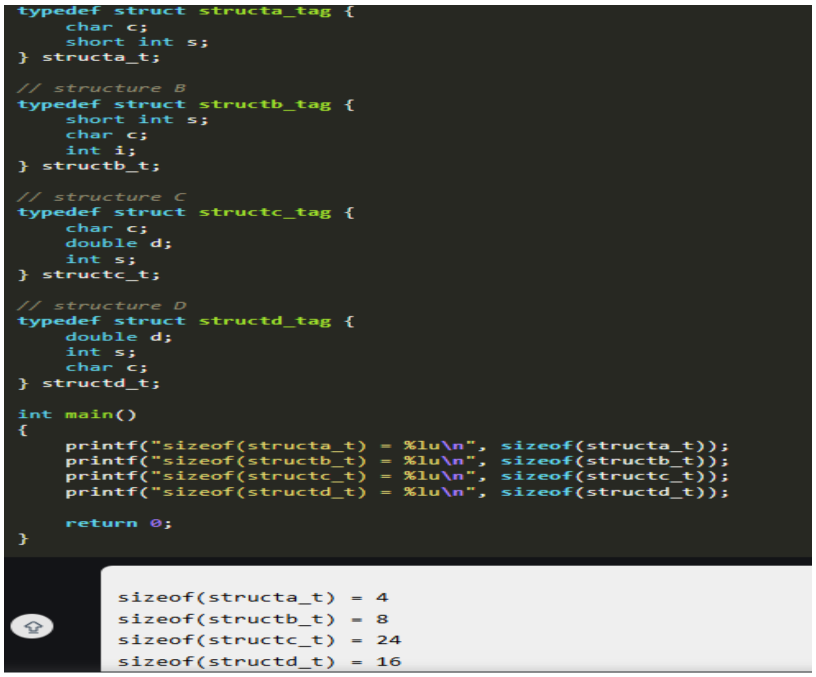
a structure is a user-defined data type that can be used to group items of possibly different types into a single type. The struct keyword is used to define a structure. The items in the structure are called its member and they can be of any valid data type.

Data Alignment in Memory:

Every data type in C will have alignment requirements (in fact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32-bit machine, the processing word size will be 4 bytes.

Structure Padding in C:

Structure padding is the addition of some empty bytes of memory in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.



```c
typedef struct structa_tag {
    char c;
    short int s;
} structa_t;

// structure B
typedef struct structb_tag {
    short int s;
    char c;
    int i;
} structb_t;

// structure C
typedef struct structc_tag {
    char c;
    double d;
    int s;
} structc_t;

// structure D
typedef struct structd_tag {
    double d;
    int s;
    char c;
} structd_t;

int main()
{
    printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Unaligned Memory:

Definition: Memory is **unaligned** when data is stored at addresses that are not multiples of the data type's size.

Characteristics:

- **No padding**, so memory is compact.
- May cause **performance penalties** due to extra CPU cycles.
- Can cause **hardware exceptions** on architectures like ARM or SPARC.
- Useful when working with **binary file formats** or **network packets**.

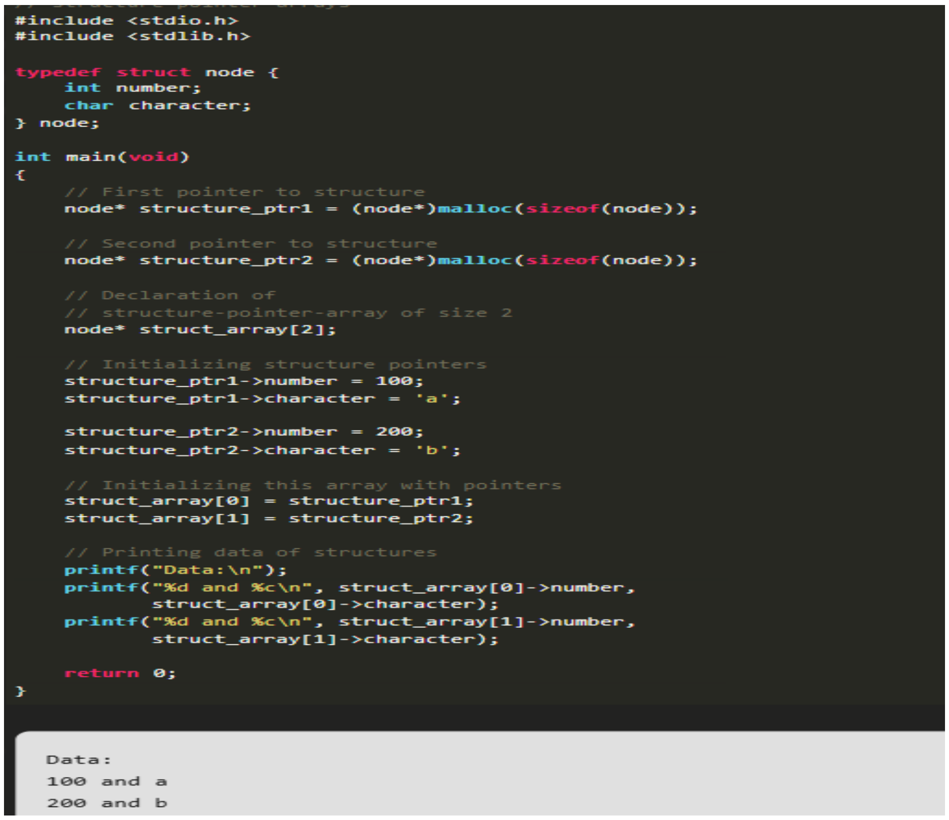
Creating structure pointer arrays (Static Arrays):

i). 1D Arrays

We can statically allocate memory for 1D and 2D arrays in C language. The static memory is allocated in the stack memory. We can do static memory allocation of structure pointers in the following ways:

Step 1 – Declaring and initializing 1D arrays

Let's say we have a structure "node", and we created different pointers to this structure. Now, to make a 1D array of those pointers in static memory, we must follow the following syntax:

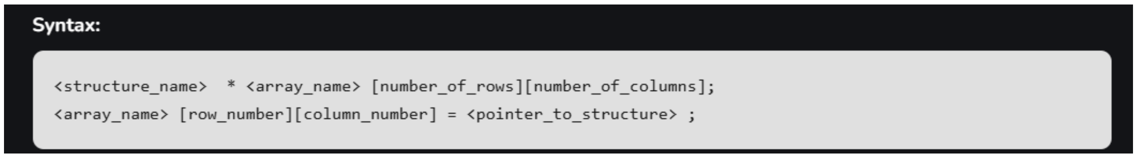


```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int number;
    char character;
} node;

int main(void)
{
    // First pointer to structure
    node* structure_ptr1 = (node*)malloc(sizeof(node));

    // Second pointer to structure
    node* structure_ptr2 = (node*)malloc(sizeof(node));

    // Declaration of
    // structure-pointer-array of size 2
    node* struct_array[2];

    // Initializing structure pointers
    structure_ptr1->number = 100;
    structure_ptr1->character = 'a';

    structure_ptr2->number = 200;
    structure_ptr2->character = 'b';

    // Initializing this array with pointers
    struct_array[0] = structure_ptr1;
    struct_array[1] = structure_ptr2;

    // Printing data of structures
    printf("Data:\n");
    printf("%d and %c\n", struct_array[0]->number,
           struct_array[0]->character);
    printf("%d and %c\n", struct_array[1]->number,
           struct_array[1]->character);

    return 0;
}
```

```
Data:
100 and a
200 and b
```

ii). 2D Arrays

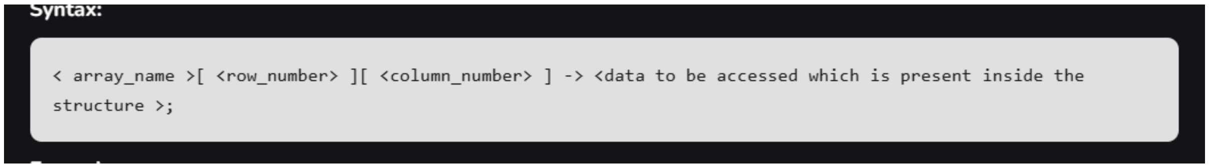
Step 1 – Declaring and initializing 2D arrays:



Syntax:

```
<structure_name>  * <array_name> [number_of_rows][number_of_columns];
<array_name> [row_number][column_number] = <pointer_to_structure> ;
```

Step 2- Accessing elements of our 2D array:



Syntax:

```
< array_name >[ <row_number> ][ <column_number> ] -> <data to be accessed which is present inside the
structure >;
```

Creating structure pointer arrays (Dynamic Arrays):

i). 1D Arrays

As we know that in C language, we can also dynamically allocate memory for our variables or arrays. The dynamically allocated variables or arrays are stored in **Heap.** To dynamically allocate memory for structure pointer arrays, one must follow the following syntax:

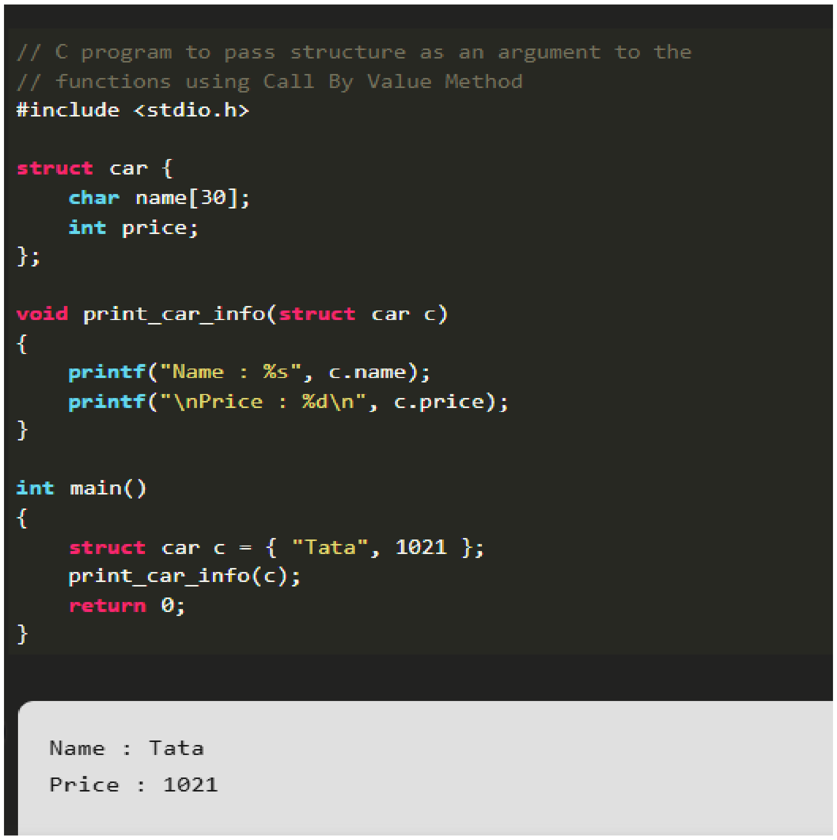
< structure_name > ** < array_name > = <structure_name **> malloc (sizeof(<structure_name>)* size) ;

How to Pass a structure as an argument to the functions?

When passing structures to or from functions in C, it is important to keep in mind that the entire structure will be copied. This can be expensive in terms of both time and memory, especially for large structures. The passing of structure to the function can be done in two ways:

- By passing all the elements to the function individually.
- By passing the entire structure to the function.

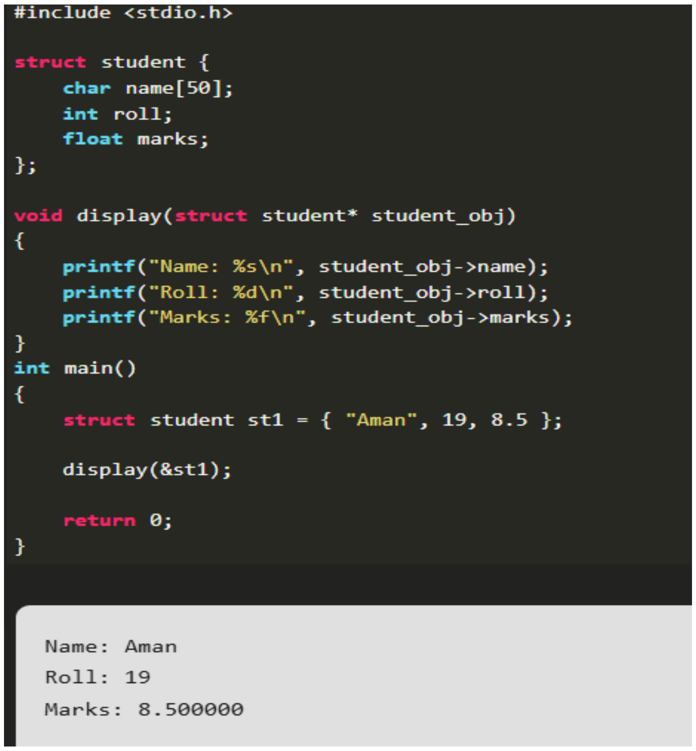
Example 1: Using Call By Value Method:



```c
// C program to pass structure as an argument to the
// functions using Call By Value Method
#include <stdio.h>

struct car {
    char name[30];
    int price;
};

void print_car_info(struct car c)
{
    printf("Name : %s", c.name);
    printf("\nPrice : %d\n", c.price);
}

int main()
{
    struct car c = { "Tata", 1021 };
    print_car_info(c);
    return 0;
}
```

```
Name : Tata
Price : 1021
```

Example 2: Using Call By Reference Method:

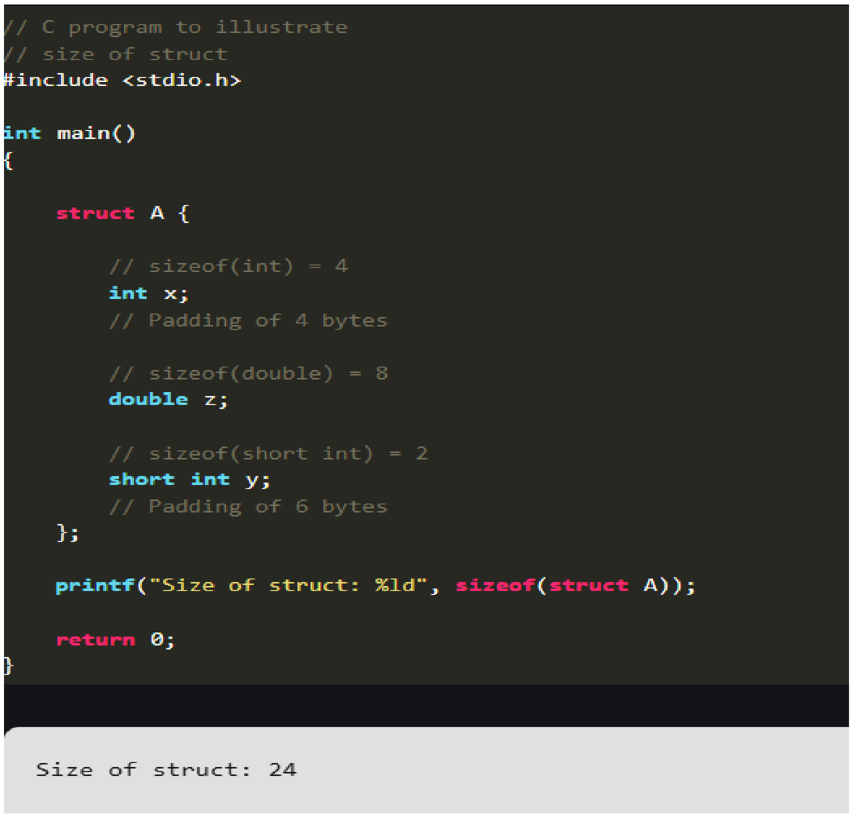


```c
#include <stdio.h>

struct student {
    char name[50];
    int roll;
    float marks;
};

void display(struct student* student_obj)
{
    printf("Name: %s\n", student_obj->name);
    printf("Roll: %d\n", student_obj->roll);
    printf("Marks: %f\n", student_obj->marks);
}
int main()
{
    struct student st1 = { "Aman", 19, 8.5 };

    display(&st1);

    return 0;
}
```

```
Name: Aman
Roll: 19
Marks: 8.500000
```

Prerequisite : sizeof operator in C
The sizeof for a struct is not always equal to the sum of sizeof of each individual member. This is because of the padding added by the compiler to avoid alignment issues. Padding is only added when a structure member is followed by a member with a larger size or at the end of the structure.

Different compilers might have different alignment constraints as C standards state that alignment of structure totally depends on the implementation.



```c
// C program to illustrate
// size of struct
#include <stdio.h>

int main()
{

    struct A {

        // sizeof(int) = 4
        int x;
        // Padding of 4 bytes

        // sizeof(double) = 8
        double z;

        // sizeof(short int) = 2
        short int y;
        // Padding of 6 bytes
    };

    printf("Size of struct: %ld", sizeof(struct A));

    return 0;
}
```

```
Size of struct: 24
```

1. **Structured Programming :**
   Structured Programming, as name suggests, is a technique that is considered as precursor to OOP and usually consists of well-structured and separated modules. In this programming, user can create its own user-defined functions as well as this methodology tries to resolve issues that are associated with unconditional transfers to allow programmers follow logic of programs. It also requires more discipline at the design and logical structuring stage.

2. **Object-Oriented Programming:**

Object-Oriented Programming ,as name suggests, is a different approach to programming that brings together data and functions that execute on them. It basically supports encapsulation, abstraction, inheritance, polymorphism, etc. It also includes data hiding feature therefore it is more secure. This model is based on real life entities that focuses on by whom task is to be done rather than focusing on what to do.