**Chapter 3 – CTSD**

**Enumerators:**

An **enum** is a special type that represents a group of constants (unchangeable values).

**OR**

In C programming, an enumeration type (also called enum) is a data type that consists of integral constants. To define enums, the **enum** keyword is used.

Syntax: **enum** enum\_name {const1, const2,…....constn};

By default, const1 is **0**, const2 is **1** and so on. You can change default values of enum elements during declaration (if necessary).

Ex: **enum** Directions{

North=0,

South=20,

East=5,

West=10

};

**Enumerator Types:**

An ***enumerated type*** is a type whose legal values consist of a fixed set of constants. Common examples include compass directions, which take the values North, South, East and West and days of the week, which take the values Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday.

***Enumerated type declaration***

As we know that in C language, we need to declare the variable of a pre-defined type such as int, float, char, etc. Similarly, we can declare the variable of a user-defined data type, such as **enum.**

Suppose we can create an enum of type Boolean as shown below:

**enum** Boolean{ false, true };

**enum** Boolean check; // declaring an enum variable check

Here, a variable **check**of the type **enum Boolean** is created.

You can also declare enum variables like this:

**enum** Boolean{ false, true }check;

Here, the value of false is equal to **0** and the value of true is equal to **1**.

Example 1:

#include <stdio.h>

**enum** week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

**void** main()

{

**enum** week today; // creating today variable of  enum week type

 today=Wednesday; // assigning value of Wednesday to w.

 printf("The value of today is %d", today + 1); // **Output: The value of today is 4 (3+1).**

}

Example 2:

#include <stdio.h>

**enum** days{sunday=1, monday, tuesday, wednesday, thursday, friday, saturday};

void main()

{

**enum**days d;

   d=monday;

   switch(d)

   {

       case sunday:

       printf("Today is sunday");

       break;

       case monday:

       printf("Today is monday");

       break;

       case tuesday:

       printf("Today is tuesday");

       break;

       case wednesday:

       printf("Today is wednesday");

       break;

       case thursday:

       printf("Today is thursday");

       break;

       case friday:

       printf("Today is friday");

       break;

       case saturday:

       printf("Today is saturday");

       break;

   }

}

**Output: Today is Monday**

Example 3:

#include <stdio.h>

**enum** year{Jan=1, Feb, Mar, Apr=8, May, Jun, Jul, Aug, Sep=2, Oct, Nov, Dec};

void main()

{

 int i;

 for (i=Jan; i<=Dec; i++)

printf("%d ", i);// **Output: 1,2,3,8,9,10,11,12,2,3,4,5**

 }

**Note:**

1) Two enums can have same value.

Ex: In the above program both Feb and Sep have same value 2.

2) If we do not explicitly assign values to enum names, the compiler by default assigns values starting from **0.**

3) We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

4) The value assigned to enum names must be some integral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

**Structures:**

A Structure is a collection of dissimilar data types (attributes or members or elements or fields) of a particular entity.

**OR**

Structure in c is a user-defined data type that enables us to store the collection of different data types.

**Declaration, Initialization and Accessing Structures:**

In C, there are cases where we need to store multiple attributes of an entity. It is not necessary that an entity has all the information of one type only. It can have different attributes of different data types. For example, an entity **Book** may have its book\_name (string), book\_pages (int), book\_price (float) etc. To store such type of information regarding an entity Book we have 2 approaches:

a) Construct individual arrays for storing book\_name, no\_of\_pages, book\_price. (or)

b) Use a special data structure called Structure to store the collection of different data types as follows:

A Structure can be declared using **struct** keyword in C.

**struct** Book

{

int book\_pages;

float book\_price;

char book\_name[20];

};

Example:

Let's see the example to define a structure for an entity employee in c.

**struct** employee

{  **int** id;   //Here id, name and salary are attributes (or)

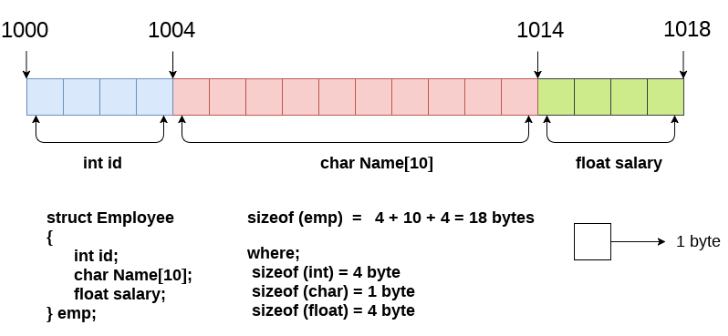
**char** name[10];  //members (or) elements (or) fields of the

**float** salary;   // structure employee

};

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields or attributes or elements of the structure.

The following image shows the memory allocation of the structure employee that is defined in the above example.



Note: Contiguous (adjacent) memory locations are used to store structure members (attributes) in the memory.

**Declaring a Structure variable:**

We can declare a variable for the structure so that we can access the member of the structure easily. There are **two** ways to declare structure variable:

1) By struct keyword within main() function

Ex:

#include<stdio.h>

struct Book

{ int pages;

float price;

char bookname[20];

};

void main()

{

struct Book b1,b2,b3; //Here b1,b2,b3 are structure Book variables

}

The variables b1, b2 and b3 can be used to access the members of the structure Book. Here, b1, b2 and b3 can be treated in the same way as the objects in [C++](https://www.javatpoint.com/cpp-tutorial) and [Java](https://www.javatpoint.com/java-tutorial). If numbers of variables are not fixed, use this approach. It provides you the flexibility to declare the structure variable many times.

2) By declaring a variable at the time of defining the structure.

Ex:

struct Book

{ int pages;

float price;

char bookname[20];

}b1,b2,b3;

If number of variables are fixed, use this approach. It saves your code to declare a variable in main() function.

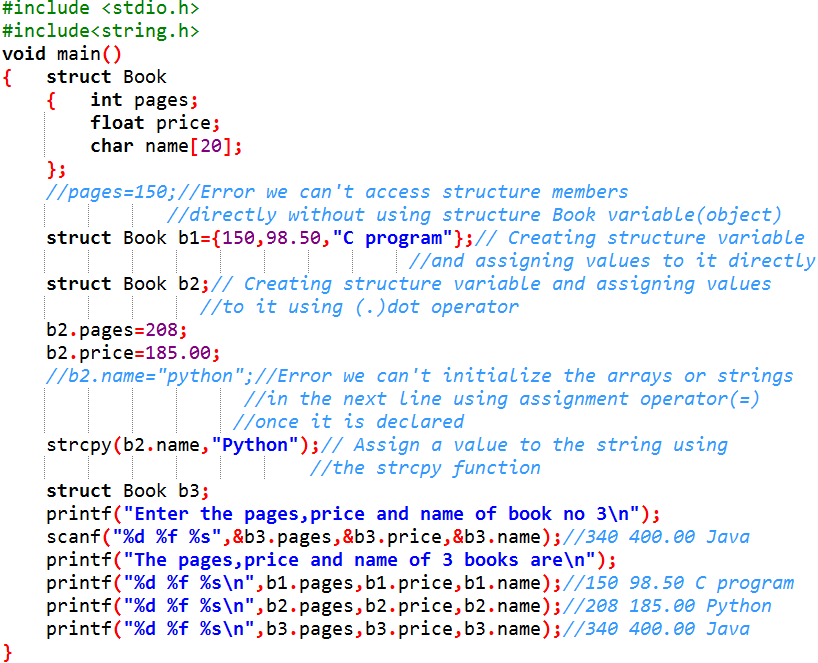
**Accessing members of the structure:**

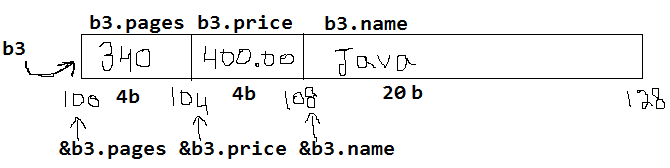
There are two ways to access structure members:

1) **. (dot operator)**  is used to access the structure members using structure variable.

2) **-> (arrow operator)** is used to access structure members using structure pointer variable.

Example programs No 1:

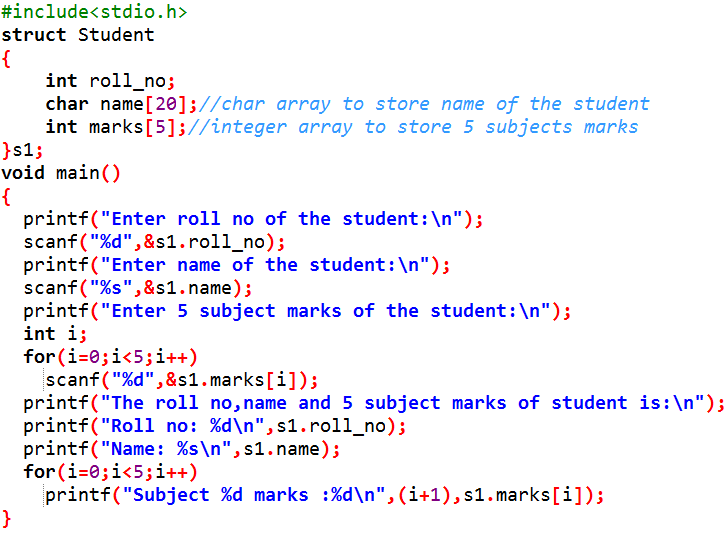




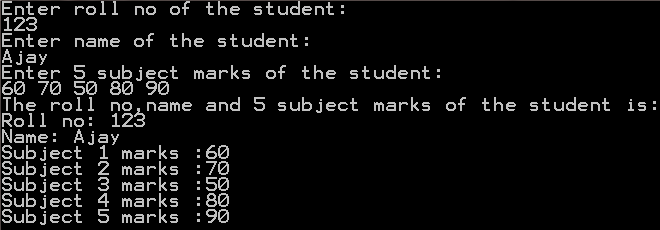
**Arrays within structures:**

An array can be declared inside a [structure](https://www.geeksforgeeks.org/structures-c/)as a member when we need to store multiple members of the same type.

Example programs No 2:



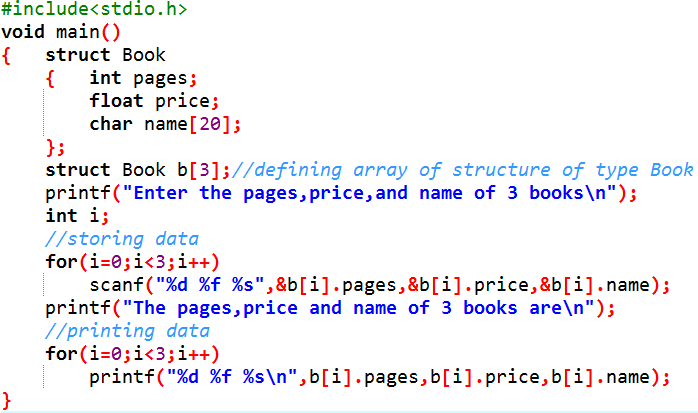
Output:

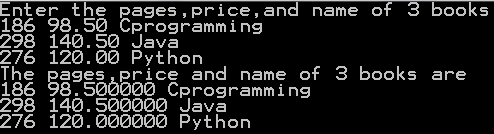


**Array of structures:**

An array whose elements are of type structure is called array of structure. It is generally useful when we need multiple structure variables in our program.

Example programs No 3:





**Nested Structures:**

A nested structure in C is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.

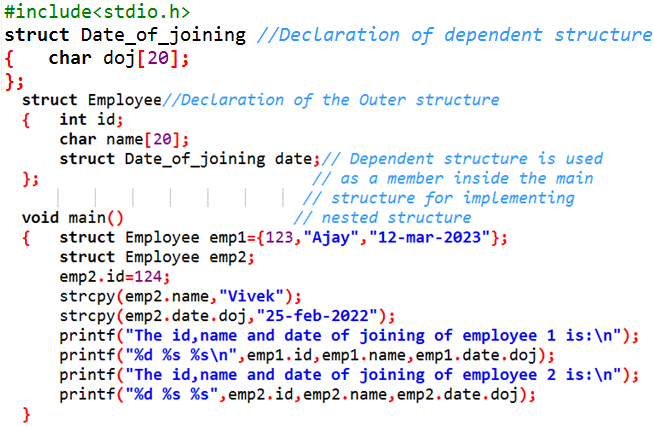
The structure can be nested in the following different ways:

1. By separate nested structure.
2. By embedded nested structure.

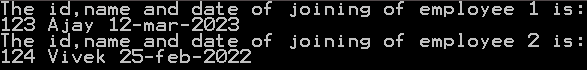
**1) By separate nested structure:**

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

Example programs No 4:



Output:



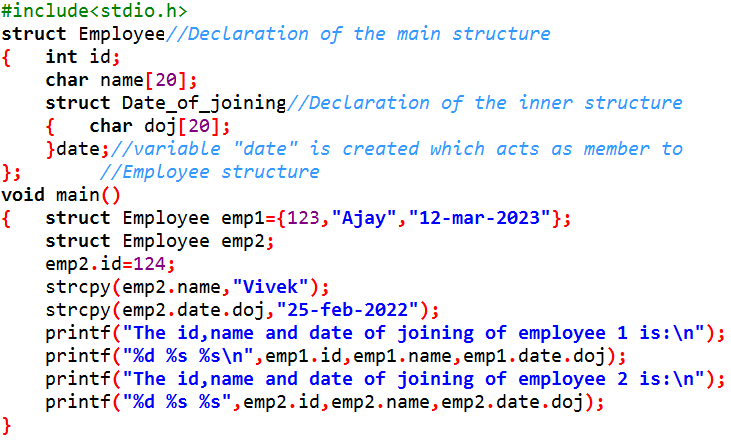
**2) By embedded nested structure:**

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it cannot be used in multiple data structures. Consider the following example.

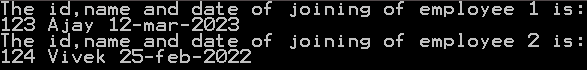
Note:

Whenever an embedded nested structure is created, the variable declaration is compulsory at the end of the inner structure, which acts as a member of the outer structure. It is compulsory that the **structure variable** is created at the end of the inner structure.

Example programs No 5:



Output:



**Pointer in Structures (Structure Pointers):**

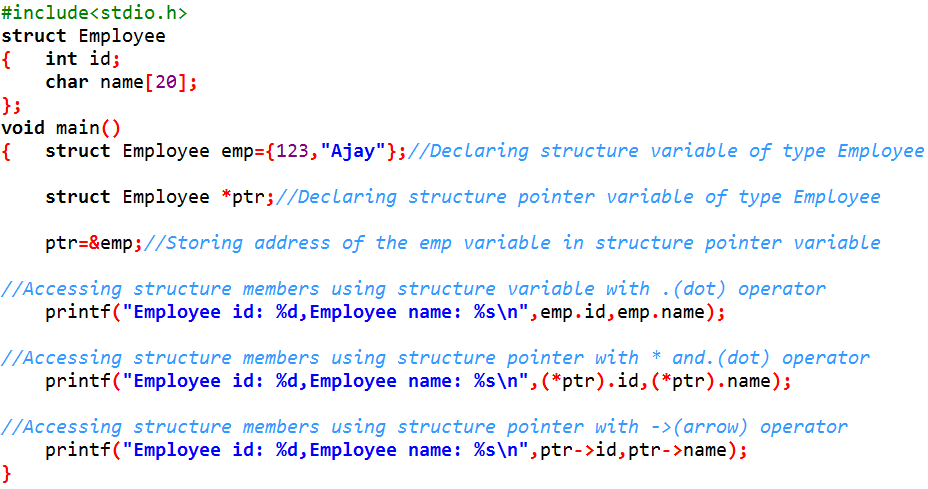
The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have pointer pointing to a struct, such pointers are known as structure pointer.

A structure pointer is defined as the [pointer](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) which points to the address of the memory block that stores a [structure](https://www.geeksforgeeks.org/structures-c/) known as the structure pointer. Complex data structures like linked lists, trees, graphs, etc. are created with the help of structure pointers. The structure pointer tells the address of a structure in memory by pointing its variable to the structure variable.

There are two ways to access the members of the structure with the help of a structure pointer:

1. With the help of (\*) asterisk or indirection operator and (.) dot operator.
2. With the help of (->) Arrow operator.

Example programs No 6:



Output:



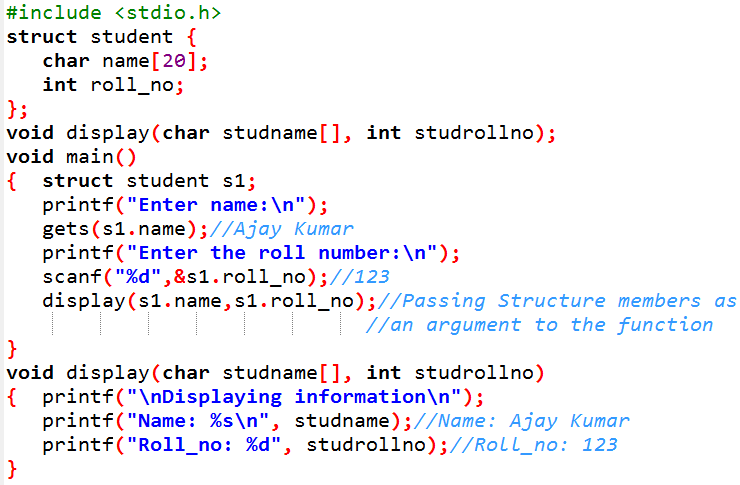
**Structure and Functions:**

Note: When a function is called, if we pass the values of the variables to the function, it is known as the **call by value**. Instead of passing the values, if we pass the address of the variables to the function, it is known as **call by reference**.

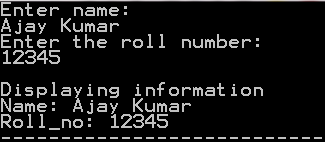
The **dot (.)** operator is used to access a structure member. The **arrow (->)** operator is to access the members of a structure when the pointer references the structure.

**Passing Structure members as an argument to the functions:**

Example programs No 7:

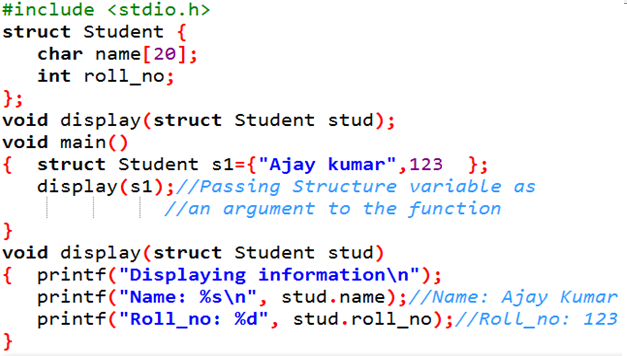


Output:



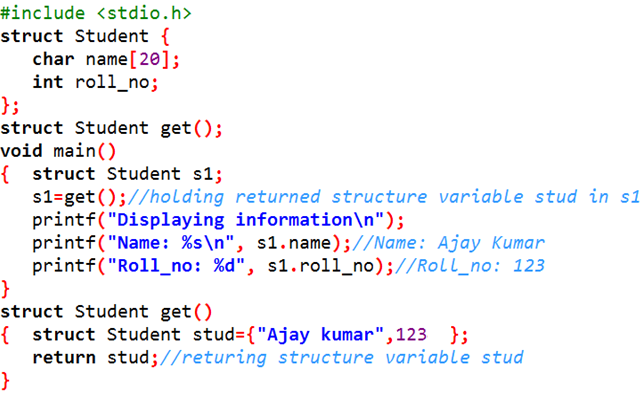
**Passing Structure variable as an argument to the functions:**

Example programs No 8:



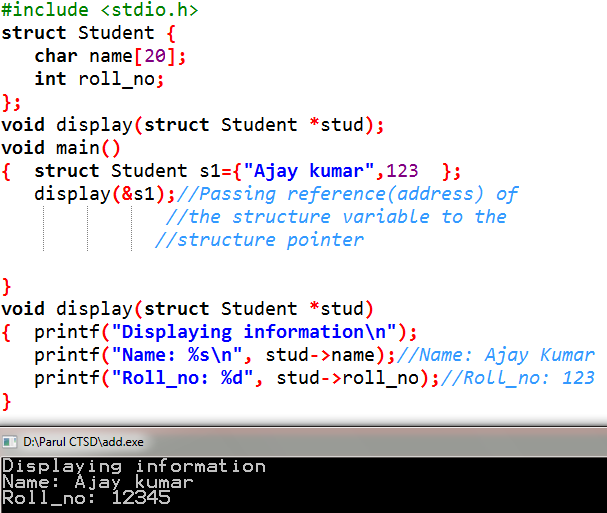
**Returning a Structure variable from the functions:**

Example programs No 9:



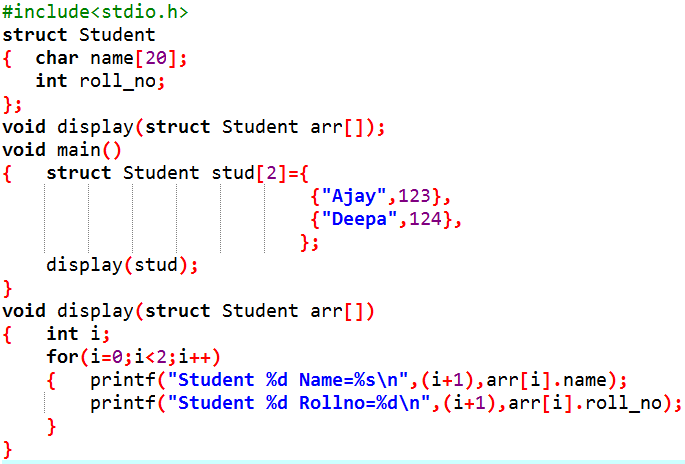
**Passing Structure pointers as an argument to the functions (or) Passing structure by reference to the functions:**

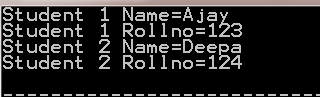
Example programs No 10:



**Passing Array of Structure as an argument to the functions:**

Example programs No 11:





**Anonymous Structures:**

Anonymous structures in C allow us to define structures without naming them, which is particularly useful in nested structures or unions, where naming the inner structure isn’t necessary.

Ex:

**struct //**Anonymous structure without any name

{

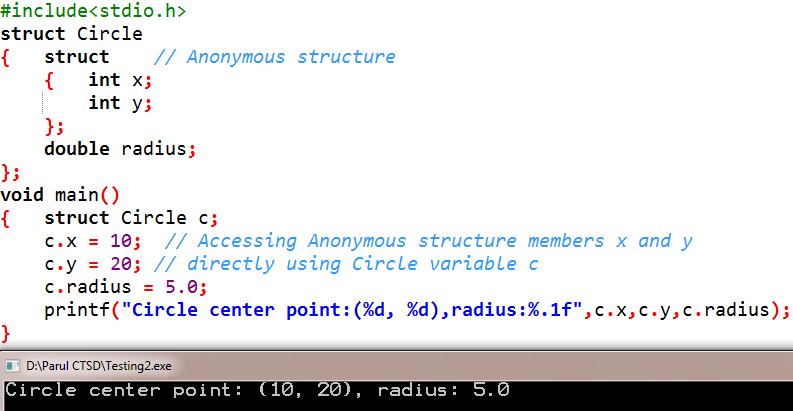
int a;

float b;

char c;

};

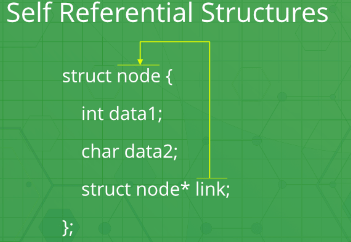
Example programs No 12:



**Self-referential structures:**

In C, a self-referential structure is a structure that contains a pointer to a variable of the same type. This allows the structure to refer to itself, creating a linked data structure.

Ex:

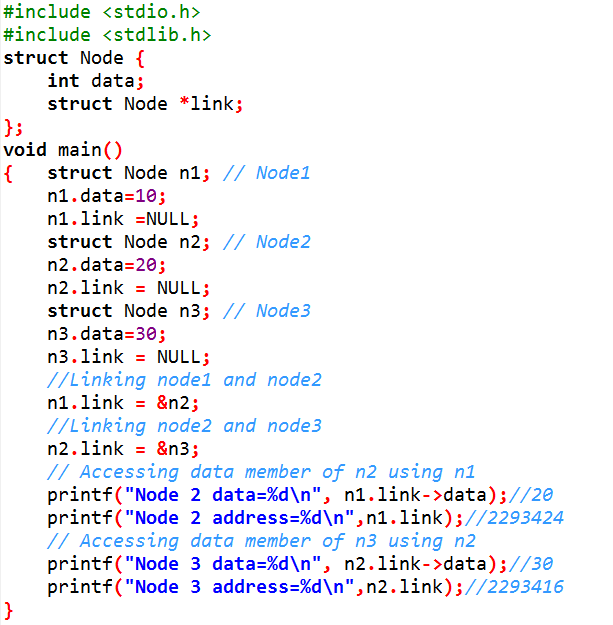


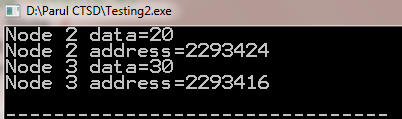
In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

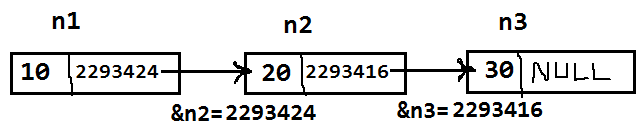
**Applications:** Self-referential structures are very useful in creation of other complex data structures like:

* [Linked Lists](https://www.geeksforgeeks.org/data-structures/linked-list/)
* [Stacks](https://www.geeksforgeeks.org/stack-data-structure/)
* [Queues](https://www.geeksforgeeks.org/queue-data-structure/)
* [Trees](https://www.geeksforgeeks.org/binary-tree-data-structure/)
* [Graphs](https://www.geeksforgeeks.org/graph-and-its-representations/) etc.

Example programs No 13:



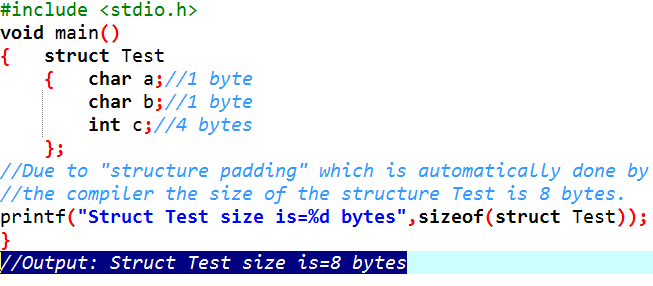




**Structure Padding:**

Structure padding is a concept in C that adds the one or more empty bytes between the memory addresses to align the data in memory.

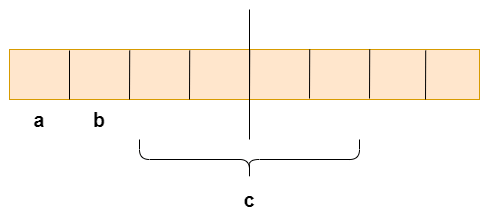
Example programs No 14:



As per the C concepts, Contiguous memory will be allocated to the structure members. In the above program **char** takes **1** byte each and **int** takes **4** bytes in memory. So only 6(1+1+4) bytes should be allocated for the above structure. But this answer is wrong. Now, we will understand why this answer is wrong? We need to understand the concept of structure padding.

Architecture of a computer processor is such a way that it can read 1 word from memory at a time.1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.

If we have a 32-bit processor (4 bytes at a time), then the pictorial representation of the memory for the above structure would be:



As we know that structure occupies the contiguous block of memory as shown in the above diagram, i.e., 1 byte for char a, 1 byte for char b, and 4 bytes for int c, then what problem do we face in this case?

### What's the problem?

The 4-bytes can be accessed at a time as we are considering the 32-bit architecture. The problem is that in one CPU cycle, one byte of **char a**, one byte of **char b**, and 2 bytes of **int c** can be accessed. We will not face any problem while accessing the **char a** and **char b** as both the variables can be accessed in one [CPU](https://www.javatpoint.com/cpu-full-form) cycle, but we will face the problem when we access the **int c** variable as 2 CPU cycles are required to access the value of the 'c' variable. In the first CPU cycle, the first two bytes are accessed, and in the second cycle, the other two bytes are accessed.

Suppose we do not want to access the 'a' and 'b' variable, we only want to access the variable 'c', which requires two cycles. The variable 'c' is of 4 bytes, so it can be accessed in one cycle also, but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles. The structure padding is done automatically by the compiler. Now, we will see how structure padding is done.

### How is structure padding done?

### 

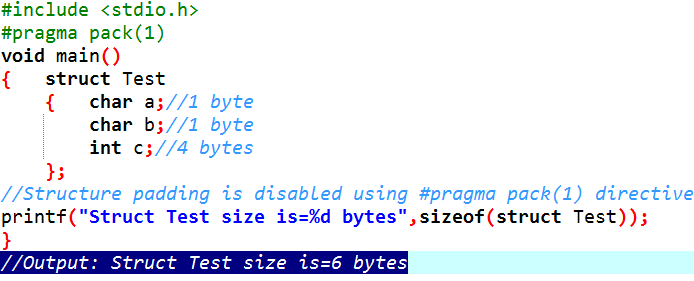
In order to achieve the structure padding, an empty row is created on the left, as shown in the above diagram, and the two bytes which are occupied by the 'c' variable on the left are shifted to the right. So, all the four bytes of 'c' variable are on the right. Now, the 'c' variable can be accessed in a single CPU cycle. After structure padding, the total memory occupied by the structure is 8 bytes (1 byte+1 byte+2 bytes+4 bytes), which is greater than the previous one. Although the memory is wasted in this case, the variable can be accessed within a single cycle.

### How to avoid the structure padding in C?

The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

We can avoid structure padding in C by using **#pragma pack(1)** directive as follows:

Example programs No 15:



**Unions:**

The **Union** is a user-defined data type in C language that can contain elements of the different data types just like **structure**. But unlike structures, all the members in the C union are stored in the same memory location. **Due to this, only one member can store data at the given instance.**

Example:

Let's see the example to define a union for an entity employee in c.

**union** employee

{  **int** id;   //Here id, name and salary are attributes (or)

**char** name[10];  //members (or) elements (or) fields of the

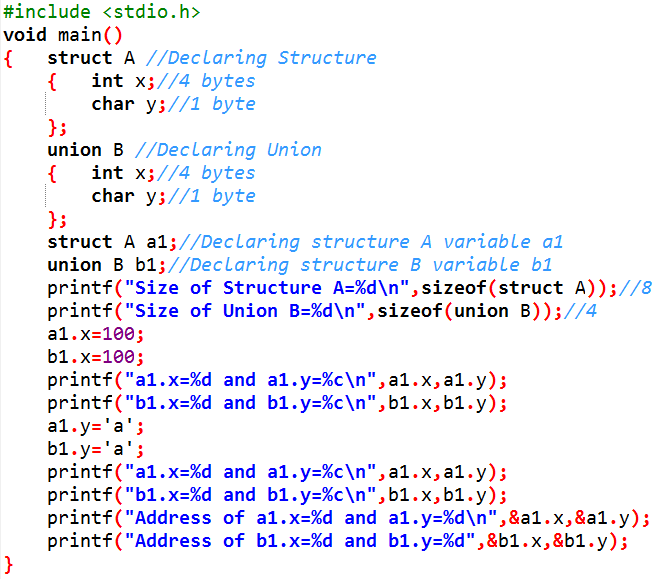
**float** salary;   // union employee

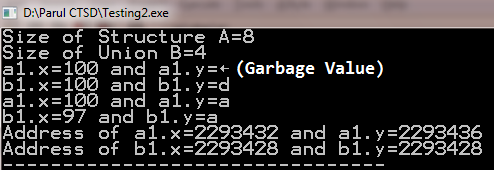
};

Here, **union** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields or attributes or elements of the union.

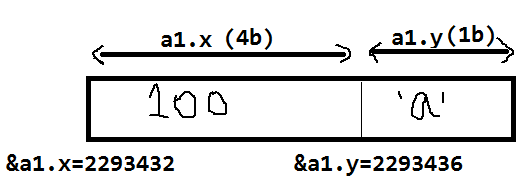
**Difference between Structure and Union:**

Example programs No 16:

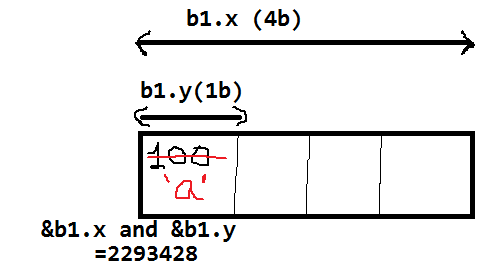




Memory representation of Structure A:



Memory representation of Union B:

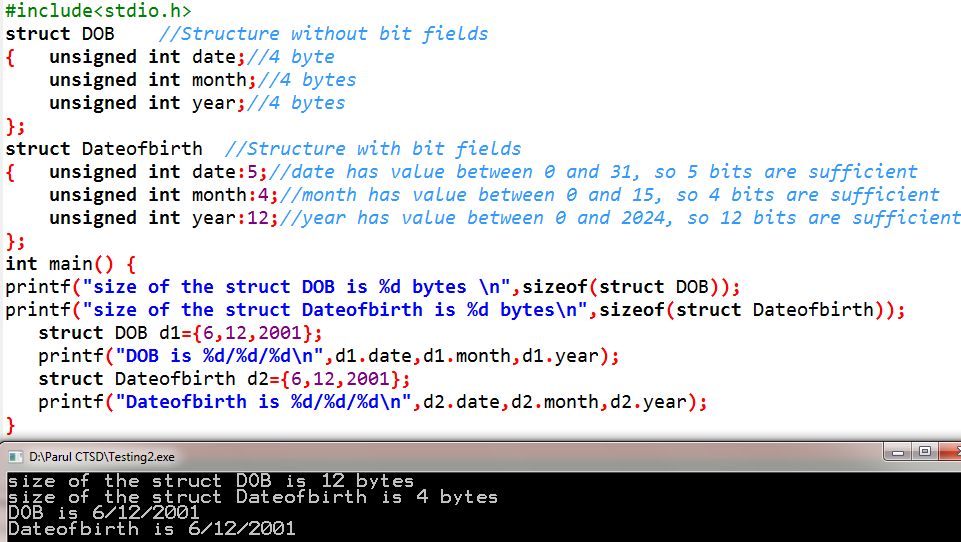


In the above example, In case of Unions, Unions share the same memory location for different member variables. Due to this, only one member can store data at the given instance. So the character associated with ASCII(American standard code for information interchange) value **100** gets printed**(‘d’)** and the ASCII value associated with the character **‘a’** gets printed**(97).**

**Bit Fields:**

In C, we can specify the size (**in bits**) of the structure and union members. The idea of bit-field is to use memory efficiently when we know that the value of a field (member) or group of fields will never exceed a limit or is within a small range. C Bit fields are used when the storage of our program is limited.

Example programs No 17:

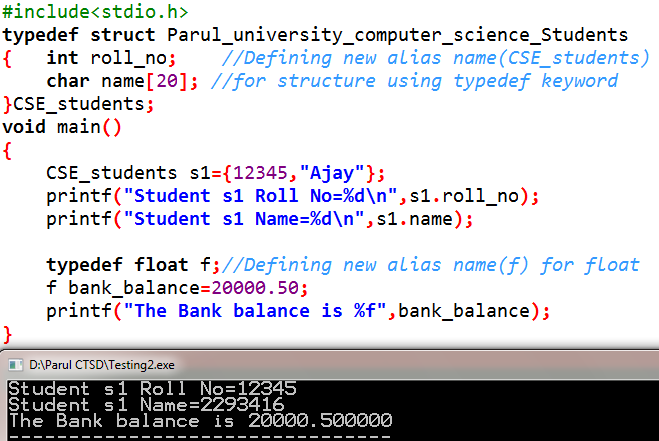


**Typedef:**

The**typedef**is a keyword that is used to provide existing data types with a new name. The C typedef keyword is used to redefine the name of already existing data types.

When names of data types become difficult to use in programs, typedef is used with user-defined data types, which behave similarly to defining an alias for commands.

Example programs No 18:



================================ END==================================