Structures in C

***What is a structure?***
A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.



***How to create a structure?***
‘struct’ keyword is used to create a structure. Following is an example.

|  |
| --- |
| struct address {    char name[50];    char street[100];    char city[50];    char state[20];    int pin; }; |

***How to declare structure variables?***
A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

|  |
| --- |
| // A variable declaration with structure declaration. struct Point {    int x, y; } p1;  // The variable p1 is declared with 'Point'     // A variable declaration like basic data types struct Point {    int x, y; };    int main() {    struct Point p1;  // The variable p1 is declared like a normal variable } |

Note: In C++, the struct keyword is optional before in declaration of a variable. In C, it is mandatory.

***How to initialize structure members?***
Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

|  |
| --- |
| struct Point {    int x = 0;  // COMPILER ERROR:  cannot initialize members here    int y = 0;  // COMPILER ERROR:  cannot initialize members here };   |

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces ‘{}’. For example, following is a valid initialization.

|  |
| --- |
| struct Point {    int x, y; };    int main() {    // A valid initialization. member x gets value 0 and y    // gets value 1.  The order of declaration is followed.    struct Point p1 = {0, 1};  } |

***How to access structure elements?***
Structure members are accessed using dot (.) operator.

|  |
| --- |
| #include<stdio.h>   struct Point {    int x, y; };   int main() {    struct Point p1 = {0, 1};      // Accesing members of point p1    p1.x = 20;    printf ("x = %d, y = %d", p1.x, p1.y);      return 0; } |

**Output:**

x = 20, y = 1

**What is designated Initialization?**
Designated Initialization allows structure members to be initialized in any order. This feature has been added in [C99 standard](https://www.geeksforgeeks.org/c-programming-language-standard/%22%20%5Ct%20%22_blank).

|  |
| --- |
| #include<stdio.h>   struct Point {    int x, y, z; };   int main() {    // Examples of initializtion using designated initialization    struct Point p1 = {.y = 0, .z = 1, .x = 2};    struct Point p2 = {.x = 20};      printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);    printf ("x = %d", p2.x);    return 0; }  |

**Output:**

x = 2, y = 0, z = 1

x = 20

***What is an array of structures?***
Like other primitive data types, we can create an array of structures.

|  |
| --- |
| #include<stdio.h>   struct Point {    int x, y; };   int main() {    // Create an array of structures    struct Point arr[10];      // Access array members    arr[0].x = 10;    arr[0].y = 20;      printf("%d %d", arr[0].x, arr[0].y);    return 0; }  |

**Output:**

10 20

***What is a structure pointer?***
Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow ( -> ) operator.

|  |
| --- |
| #include<stdio.h>   struct Point {    int x, y; };   int main() {    struct Point p1 = {1, 2};      // p2 is a pointer to structure p1    struct Point \*p2 = &p1;      // Accessing structure members using structure pointer    printf("%d %d", p2->x, p2->y);    return 0; } |

**Output:**

1 2

Self Referential Structures

Self Referential structures are those [structures](https://www.geeksforgeeks.org/structures-c/) that have one or more pointers which point to the same type of structure, as their member.



In other words, structures pointing to the same type of structures are self-referential in nature.

Example:

|  |
| --- |
| struct node {     int data1;     char data2;     struct node\* link; };   int main() {     struct node ob;     return 0; }  |

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.
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

**Types of Self Referential Structures**

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

**Self Referential Structure with Single Link:**These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.


|  |
| --- |
| #include <stdio.h>   struct node {     int data1;     char data2;     struct node\* link; };   int main() {     struct node ob1; // Node1       // Intialization     ob1.link = NULL;     ob1.data1 = 10;     ob1.data2 = 20;       struct node ob2; // Node2       // Initialization     ob2.link = NULL;     ob2.data1 = 30;     ob2.data2 = 40;       // Linking ob1 and ob2     ob1.link = &ob2;       // Accessing data members of  ob2 using ob1     printf("%d", ob1.link->data1);     printf("\n%d", ob1.link->data2);     return 0; }  |

**Output:**

30

40

 **Self Referential Structure with Multiple Links:**Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.

The connections made in the above example can be understood using the following figure.


|  |
| --- |
| #include <stdio.h>   struct node {     int data;     struct node\* prev\_link;     struct node\* next\_link; };   int main() {     struct node ob1; // Node1       // Intialization     ob1.prev\_link = NULL;     ob1.next\_link = NULL;     ob1.data = 10;       struct node ob2; // Node2       // Intialization     ob2.prev\_link = NULL;     ob2.next\_link = NULL;     ob2.data = 20;       struct node ob3; // Node3       // Intialization     ob3.prev\_link = NULL;     ob3.next\_link = NULL;     ob3.data = 30;       // Forward links     ob1.next\_link = &ob2;     ob2.next\_link = &ob3;       // Backward links     ob2.prev\_link = &ob1;     ob3.prev\_link = &ob2;       // Accessing  data of ob1, ob2 and ob3 by ob1     printf("%d\t", ob1.data);     printf("%d\t", ob1.next\_link->data);     printf("%d\n", ob1.next\_link->next\_link->data);       // Accessing data of ob1, ob2 and ob3 by ob2     printf("%d\t", ob2.prev\_link->data);     printf("%d\t", ob2.data);     printf("%d\n", ob2.next\_link->data);       // Accessing data of ob1, ob2 and ob3 by ob3     printf("%d\t", ob3.prev\_link->prev\_link->data);     printf("%d\t", ob3.prev\_link->data);     printf("%d", ob3.data);     return 0; }  |

**Output:**

10 20 30

10 20 30

10 20 30

In the above example we can see that ‘ob1’, ‘ob2’ and ‘ob3’ are three objects of the self referential structure ‘node’. And they are connected using their links in such a way that any of them can easily access each other’s data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

**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