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?**
  - To define a structure, we must use the ***struct*** statement.
  - The struct statement defines a new data type, with more than one member.
  - **Syntax:**

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

- **How to Declare Structure Variables?**
  - A structure variable can either be declared with structure declaration or as a separate declaration like basic types.
  - **Example:**

```
// 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
}
```

- **How to Initialize Structure Members?**
  - Structure members *cannot be* initialized with declaration.
  - **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
};
```

Explanation: 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.

  - **Example:** The following is a valid initialization.

```
// Structure members can be initialized using curly braces '{}'.
struct Point
{
        int x, y;
};

int main()
{
        // A valid initialization. member x gets value 0 and y gets value 1.
        struct Point p1 = {0, 1};
}
```

- **How to access Structure Elements?**
  - To access any member of a structure, we use the **member access operator (.)**.
  - The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.
  - We would use the keyword **struct** to define variables of structure type.
  - **Example:**
    - *Input:*

```
struct Point
{
        int x, y;
};

int main()
{
        struct Point p1 = {0, 1};
        // Accessing members of point p1
        p1.x = 20;
        printf ("x = %d, y = %d", p1.x, p1.y);
}
```

- *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.
  - This feature is not available in C++ and works only in C.
  - **Example:**
    - *Input:*

```
struct Point
{
        int x, y, z;
};

int main()
{
        // Examples of initialization 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);
}
```

- *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.
  - **Example:**
    - *Input:*

```
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);
	}
```

- *Output:*

```
	10 20
```

- **What is an Array within Structures?**
  - Arrays may be the member within structure, this is known as arrays within structure. Accessing arrays within structure is similar to accessing other members.
  - **Example:**
    - *Input:*

```
	int main()
	{
		struct student
		{
			char name[30];
			int rollno;
		} stud;

		printf ("Enter your RollNo : ");
		scanf ("%d",&stud.rollno);
		printf ("\nEnter your Name : ");
		scanf ("%s", stud.name);

		printf ("RollNo : %d\n, stud.rollno);
		printf (“Name : %s\n", stud.name);
	}
```

- *Output*

```
        Enter your RollNo : 16
        Enter your Name : Jiju
        RollNo : 16
        Name : Jiju
```

- **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.
  - **Example:**

```
    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);
    }
```

- **What is a Nested Structure?**
  - Nested structure in C is nothing but structure within structure.
  - One structure can be declared inside other structure as we declare structure members inside a structure.
  - The structure variables can be a normal structure variable or a pointer variable to access the data.
  - **Example:**
    - *Input:*

```
        struct student_college_detail
        {
                int college_id;
                char college_name[50];
        };

        struct student_detail
        {
                int id;
                char name[20];
                float percentage;
                // structure within structure
                struct student_college_detail clg_data;
        }stu_data;

        int main()
        {
                 struct student_detail stu_data = {149, "Vartika", 70, 132, "GNIT"};
                 printf(" Id is: %d \n", stu_data.id);
                 printf(" Name is: %s \n", stu_data.name);
                printf(" Percentage is: %f \n\n", stu_data.percentage);

                printf(" College Id is: %d \n", stu_data.clg_data.college_id);
                 printf(" College Name is: %s \n", stu_data.clg_data.college_name);
        }
```

- *Output:*

```
Id is: 149
Name is: Vartika
Percentage is: 70.0000
College Id is: 132
College Name is: GNIT
```

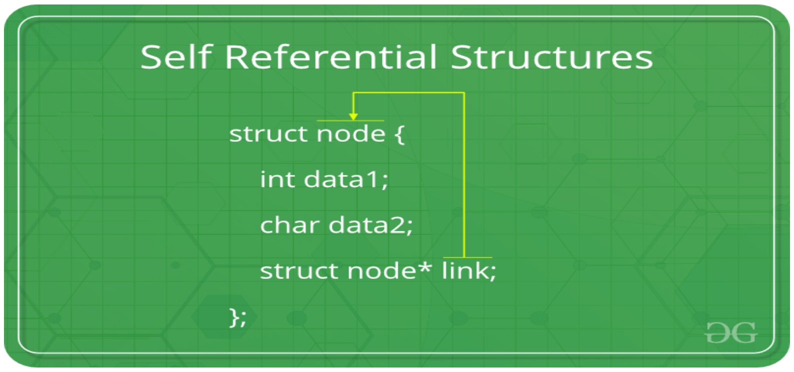
- **What is Self Referential Structures?**
  - Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.
  - In other words, structures pointing to the same type of structures are self-referential in nature.
  - **Example:**

```
struct node
{
        int data1;
        char data2;
        struct node* link;
};

int main()
{
        struct node ob;
}
```

Explanation: 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.

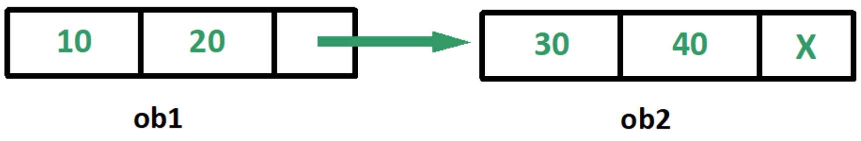


Self Referential Structures

```
struct node {
  int data1;
  char data2;
  struct node* link;
};
```

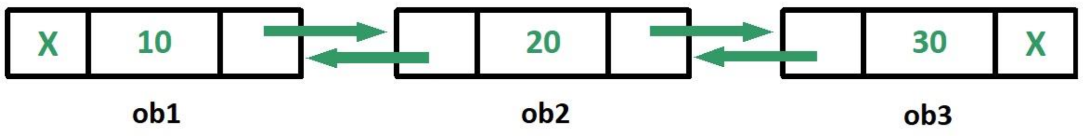
- **Types of Self Referential Structures**

1. Self Referential Structure with Single Link:
    - These structures can have only one self-pointer as their member.
    - The objects of a self-referential structure with the single link and access the corresponding data members.



2. Self Referential Structure with Multiple Link:
    - 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 diagram shows one such structure with more than one links.



- **Applications:** Self Referential structures are very useful in creation of other complex data structures like:
  - Linked Lists
  - Stacks
  - Queues
  - Trees
  - Graphs etc

- **What is Structure Padding?**
  - In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called **structure padding**.
  - Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
  - To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member's address.
  - Because of this structure padding concept in C, size of the structure is always not same as what we think.
  - Padding increases the performance of the processor at the penalty of memory. In structure or union data members aligned as per the size of the highest bytes member to prevent the penalty of performance.

- **What is Structure Packing?**
  - In Structure, sometimes the size of the structure is more than the size of all structures members because of structure padding.
  - Packing, on the other hand prevents compiler from doing padding means remove the unallocated space allocated by structure.

- **Example:** A simple C code (Problem Code)
  - *Input:*

```
struct s
{
	int i;
	char ch;
	double d;
};

int main()
{
	struct s A;
	printf("Size of A is: %ld", sizeof(A));
}
```

  - *Output:*

```
Size of A is: 16
```

Explanation: But what actual size of all structure member is 13 Bytes. So here total 3 bytes are wasted.

- So, to avoid structure padding we can use pragma pack as well as an attribute. Below are the solutions to avoid structure padding:

- **Program-1:** *Using pragma pack*
  - *Input:*

```
// To force compiler to use 1 byte packaging
#pragma pack(1)
struct s
{
	int i;
	char ch;
	double d;
};

int main()
{
	struct s A;
	printf("Size of A is: %ld", sizeof(A));
}
```

  - *Output:*

```
Size of A is: 13
```

- **Program-2:** *Using attribute*
  - *Input:*

```
struct s
{
        int i;
        char ch;
        double d;
} __attribute__((packed));
// Attribute informing compiler to pack all members

int main()
{
        struct s A;
        printf("Size of A is: %ld", sizeof(A));
}
```

  - *Output:*

```
Size of A is: 13
```

- **What are Limitations of Structures?**
  - Structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures have some limitations.
  - The C structure does not allow the struct data type to be treated like built-in data types.
  - We cannot use operators like +, - etc. on structure variables
  - **No Data Hiding:** C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
  - **Functions inside Structure:** C structures do not permit functions inside Structure
  - **Static Members:** C Structures cannot have static members inside their body
  - **Access Modifiers:** C Programming language do not support access modifiers. So they cannot be used in C Structures.
  - **Construction creation in Structure:** Structures in C cannot have constructor inside Structures.