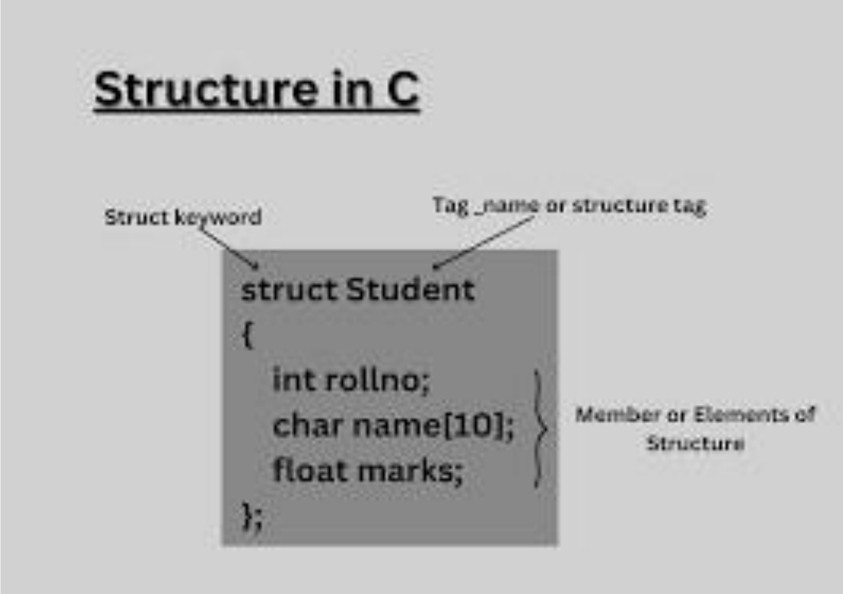
C Structures:

The structure in C 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 the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type. Additionally, the values of a structure are stored in contiguous memory locations.



C Structure Declaration

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables along with their datatype. We can use the struct keyword to declare the structure in C using the following syntax: **Syntax**

```
struct structure_name {
data_type member_name1;
data_type member_name1;
   ....
   ....
};
```

The above syntax is also called a structure template or structure prototype and no memory is allocated to the structure in the declaration

C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
struct structure_name {
data_type member_name1;
data_type member_name1;
   ....
   ....
}variable1, varaible2, ...;
```

2. Structure Variable Declaration after Structure Template

```
// structure declared beforehand
struct structure_name variable1, variable2, .......;
```

Access Structure Members

We can access structure members by using the **(.) dot operator.** **Syntax**

```
structure_name.member1; strcuture_name.member2;
```

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

Initialize Structure Members

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

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

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

Default Initialization

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

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

};

struct Point p = {0}; // Both x and y are initialized to 0
```

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator

struct *structure_name* str;
str.member1 = value1; str.member2
= value2; str.member3 = value3;
.
.
.

2. Initialization using Initializer List struct

structure_name str = { value1, value2, value3 };

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

3. Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.

struct *structure_name* str = { .member1 = value1, .member2 = value2, .member3 = value3 };

The Designated Initialization is only supported in C but not in C++.

Example of Structure in C

The following C program shows how to use structures

```
// C program to illustrate the use of structures

#include <stdio.h>


// declaring structure with name str1

struct str1 {

   int    i;

char    c;

float    f;

char

s[30];

};
```

```
// declaring structure with name str2 struct
str2 {
   int ii;
char cc;
   float ff;
} var; // variable declaration with structure template


// Driver code int
main()
{
   // variable declaration after structure template
   // initialization with initializer list and designated
   //    initializer list     struct str1 var1 = { 1, 'A', 1.00,
"GeeksforGeeks" },
             var2;     struct str2 var3 = { .ff = 5.00,
.ii = 5, .cc = 'a' };


   // copying structure using assignment operator
var2 = var1;


   printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
         var1.i, var1.c, var1.f, var1.s);     printf("Struct
2:\n\ti = %d, c = %c, f = %f, s = %s\n",
         var2.i, var2.c, var2.f, var2.s);     printf("Struct
3\n\ti = %d, c = %c, f = %f\n", var3.ii,           var3.cc,
var3.ff);
return 0;
}
```

Output

```
Struct 1:
   i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:     i = 1, c = A, f = 1.000000, s =
GeeksforGeeks Struct 3    i = 5, c = a, f =

5.000000
```

typedef for Structures

The typedef keyword is used to define an alias for the already existing datatype. In structures, we have to use the struct keyword along with the structure name to define the variables. Sometimes, this increases the length and complexity of the code. We can use the typedef to define some new shorter name for the structure.

Example

```
// C Program to illustrate the use of typedef with
// structures
#include <stdio.h>

// defining structure typedef
struct {
   int a;
} str1;

// another way of using typedef with structures typedef
struct {
   int x;
} str2;

int main()
{
   // creating structure variables using new names str1
   var1 = { 20 };

   str2 var2 = { 314 };

   printf("var1.a = %d\n", var1.a);
printf("var2.x = %d\n", var2.x);
```

```
   return 0;
}
```

Output var1.a

= 20 var2.x =

314

Nested Structures

C language allows us to insert one structure into another as a member. This process is called nesting and such structures are called nested structures. There are two ways in which we can nest one structure into another:

1. Embedded Structure Nesting

In this method, the structure being nested is also declared inside the parent structure. **Example**

```
struct parent {
int member1;
   struct member_str member2 {
      int member_str1;
char member_str2;
      ...
}    ...
}
```

2. Separate Structure Nesting

In this method, two structures are declared separately and then the member structure is nested inside the parent structure. **Example**

```
struct  member_str  {
int       member_str1;
char member_str2;
   ...

}
struct parent {
int member1;
   struct member_str member2;
   ...
}
```

One thing to note here is that the declaration of the structure should always be present before its definition as a structure member. For example, the **declaration below is invalid** as the struct mem is not defined when it is declared inside the parent structure.

```
struct parent {
   struct mem a;
};

struct mem {
   int var; };
```

Accessing Nested Members

We can access nested Members by using the same (.) dot operator two times as shown:

str_parent.str_child.member;

Example of Structure Nesting

```
// C Program to illustrate structure nesting along with
// forward declaration
#include <stdio.h>

// child structure declaration struct
child {
   int x;
char c;
};

// parent structure declaration struct
parent {
   int a;
   struct child b;
};

// driver code int
main()
{
   struct parent var1 = { 25, 195, 'A' };

   // accessing and printing nested members
printf("var1.a = %d\n", var1.a);     printf("var1.b.x
= %d\n", var1.b.x);     printf("var1.b.c = %c",

var1.b.c);


    return 0;

}
```

Output var1.a

```
= 25 var1.b.x

= 195 var1.b.c

= A
```

Structure Pointer in C

We can define a pointer that points to the structure like any other variable. Such pointers are generally called Structure Pointers. We can access the members of the structure pointed by the structure pointer using the **(->) arrow operator.**

Example of Structure Pointer

```
// C program to illustrate the structure pointer

#include <stdio.h>


// structure declaration struct

Point {

    int x, y;

};

int main()

{

    struct Point str = { 1, 2 };


    // p2 is a pointer to structure p1

struct Point* ptr = &str;


    // Accessing structure members using structure pointer

printf("%d %d", ptr->x, ptr->y);
```

```
    return 0;

}
```

Output

```
1 2
```

Self-Referential Structures

The self-referential structures in C are those structures that contain references to the same type as themselves i.e. they contain a member of the type pointer pointing to the same structure type.

Example of Self-Referential Structures

```
struct structure_name {
data_type member1;     data_type
member2;
   struct structure_name* str;
}
```

```
// C program to illustrate the self referential structures

#include <stdio.h>


// structure template

typedef struct str {

int mem1;     int

mem2;

   struct str* next;

}str;


// driver code int

main()

{     str var1 = { 1, 2, NULL };

str var2 = { 10, 20, NULL };


   // assigning the address of var2 to var1.next

var1.next = &var2;
```

```
   // pointer to var1
str *ptr1 = &var1;

   // accessing var2 members using var1     printf("var2.mem1:
%d\nvar2.mem2: %d", ptr1->next->mem1,
         ptr1->next->mem2);

   return 0;
}
```

Output

```
var2.mem1:
10 var2.mem2: 20
```

Such kinds of structures are used in different data structures such as to define the nodes of linked lists, trees, etc.

C Structure Padding and Packing

Technically, the size of the structure in C should be the sum of the sizes of its members. But it may not be true for most cases. The reason for this is Structure Padding.

Structure padding is the concept of adding multiple empty bytes 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.

There are some situations where we need to pack the structure tightly by removing the empty bytes. In such cases, we use **Structure Packing.** C language provides two ways for structure packing:

1. Using #pragma pack(1)
2. Using __attribute((packed))__

Example of Structure Padding and Packing

```
// C program to illustrate structure padding and packing
#include <stdio.h>

// structure with padding
```

```
struct str1 {
char c;    int
i;
};

struct str2 {
char c;
   int i;
} __attribute((packed)) __; // using structure packing

// driver code int
main()
{
   printf("Size of str1: %d\n", sizeof(struct str1));
printf("Size of str2: %d\n", sizeof(struct str2));    return
0;
}
```

Output

```
Size of str1: 8
Size of str2: 5
```

Bit Fields

Bit Fields are used to specify the length of the structure members in bits. When we know the maximum length of the member, we can use bit fields to specify the size and reduce memory consumption.

Syntax of Bit Fields

```
struct structure_name {
   data_type member_name: width_of_bit-field;
};
```

Example of Bit Fields

```
// C Program to illustrate bit fields in structures
#include <stdio.h>
```

```
// declaring structure for reference struct
str1 {
   int a;
char c;
};
// structure with bit fields struct str2 {
int a : 24; // size of 'a' is 3 bytes = 24 bits
char c;
};
// driver code int
main()
{
   printf("Size of Str1: %d\nSize of Str2: %d",
sizeof(struct str1), sizeof(struct str2));     return 0;
}
```

Output

```
Size of Str1: 8
Size of Str2: 4
```

Uses of Structure in C

C structures are used for the following:

1. The structure can be used to define the custom data types that can be used to create some complex data types such as dates, time, complex numbers, etc. which are not present in the language.
2. It can also be used in data organization where a large amount of data can be stored in different fields.
3. Structures are used to create data structures such as trees, linked lists, etc.
4. They can also be used for returning multiple values from a function.

Limitations of C Structures

In C language, 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 also have some limitations.

- **Higher Memory Consumption:** It is due to structure padding.

- **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 the structure so we cannot provide the associated functions.
- **Static Members:** C Structure cannot have static members inside its body.
- **Construction creation in Structure:** Structures in C cannot have a constructor inside Structures.