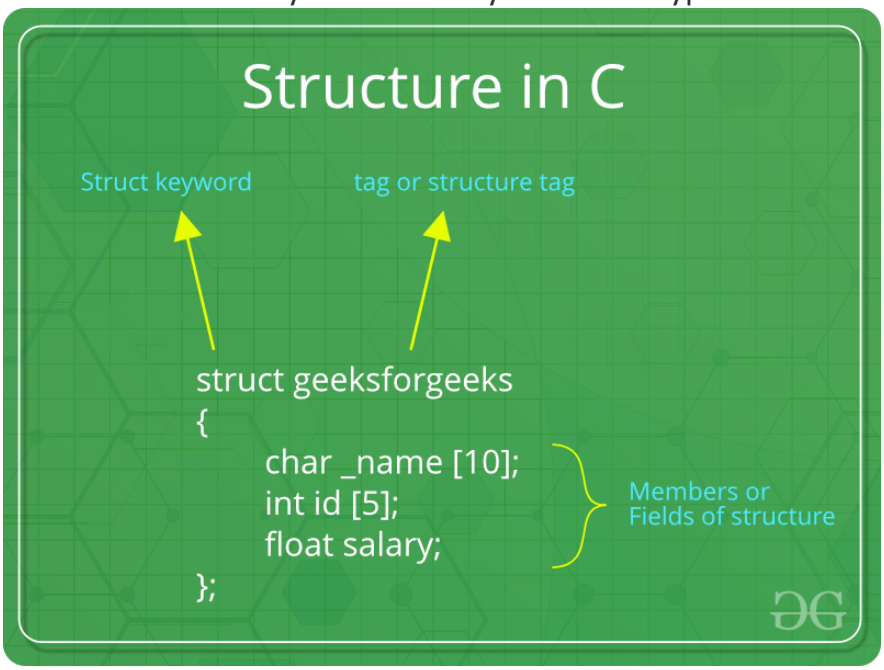
UNIT 4:

Structures, Unions and Pointers

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.



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.

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

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

```
// 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

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.

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

Syntax of union: union

student

{

datatype member1; datatype

member2;

};

Like structure variable, union variable can be declared with definition or separately such as union union

name

{

Datatype member1;

}var1;

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student struct

student

{

int i;

char ch[10];

};struct student s;

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

POINTER

A pointer is a variable that store memory address or that contains address of another variable where addresses are the location number always contains whole number. So, pointer contain always the whole number. It is called pointer because it points to a particular location in memory by storing address of that location.

Syntax-

Data type *pointer name;

Here * before pointer indicate the compiler that variable declared as a pointer. e.g.

int *p1; //pointer to integer type

float *p2; //pointer to float type char

*p3; //pointer to character type

When pointer declared, it contains garbage value i.e. it may point any value in the memory.

Two operators are used in the pointer i.e. **address operator(&)** and **indirection operator or dereference operator (*).**

Indirection operator gives the values stored at a particular address.

Address operator cannot be used in any constant or any expression.

Example:

```
void main()
{ int
i=105;
int*p;
p=&i; t
printf("value of i=%d",*p); printf("value of
i=%d",*/&i); printf("address of i=%d",&i);
printf("address of i=%d",p); printf("address of
p=%u",&p);
}
```

Pointer Expression

Pointer assignment

int i=10;

int *p=&i;//value assigning to the pointer

Here declaration tells the compiler that P will be used to store the address of integer value or in other word P is a pointer to an integer and *p reads the **value at the address contain in p.** P++;

printf("value of p=%d");

We can assign value of 1 pointer variable to other when their base type and data type is same or both the pointer points to the same variable as in the array.

Int *p1,*p2;

P1=&a[1];

P2=&a[3];

We can assign constant 0 to a pointer of any type for that symbolic constant '**NULL**' is used such as

*p=NULL;

It means pointer doesn't point to any valid memory location.

Pointer Arithmetic

Pointer arithmetic is different from ordinary arithmetic and it is perform relative to the data type(base type of a pointer).

Example:-

If integer pointer contain address of 2000 on incrementing we get address of 2002 instead of 2001, because, size of the integer is of 2 bytes.

Note:-

When we move a pointer, somewhere else in memory by incrementing or decrement or adding or subtracting integer, it is not necessary that, pointer still pointer to a variable of same data, because, memory allocation to the variable are done by the compiler.

But in case of array it is possible, since there data are stored in a consecutive manner.

Ex:void main( ) {

static int a[ ]={20,30,105,82,97,72,66,102}; int *p,*p1;

P=&a[1]; P1=&a[6];

printf("%d",*p1-*p);

printf("%d",p1-p);

}

Arithmetic operation never perform on pointer are: addition, multiplication and division of two pointer, multiplication between the pointer by any number, division of pointer by any number, add of float or double value to the pointer.

Operation performed in pointer are:

/* Addition of a number through pointer */

Example-

```
int i=100;
int *p;
p=&i;
p=p+2;
p=p+3;
p=p+9;
```

ii /* Subtraction of a number from a pointer'*/ Ex:int

```
i=22;
*p1=&a; p1=p1-10;
p1=p1-2;
```

iii- Subtraction of one pointer to another is possible when pointer variable point to an element of same type such as an array. Ex:-

```
int tar[ ]={2,3,4,5,6,7};
int *ptr1,*ptr1;

ptr1=&a[3];//2000+4
ptr2=&a[6];  //2000+6
```

Pointer vs array

```
Example :    void
main()
{
static char arr[]="Rama"; char*p="Rama";
```

printf(“%s%s”, arr, p);

In the above example, at the first time printf( ), print the same value array and pointer.

Here array arr, as **pointer to character** and **p act as a pointer to array of character .** When we are trying to increase the value of arr it would give the error because its known to compiler about an array and its base address which is always printed to base address is known as constant pointer and the base address of array which is not allowed by the compiler.

printf(“size of (p)”,size of (ar)); size of (p) 2/4 bytes

size of(ar) 5 byes