POINTERS

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Declaring a Pointer

Like variables, pointers are declared before they can be used in your program. It is declared along with an asterisk symbol (*). A pointer declaration has the following form.

Syntax:-

data_type * pointer_variable_name;

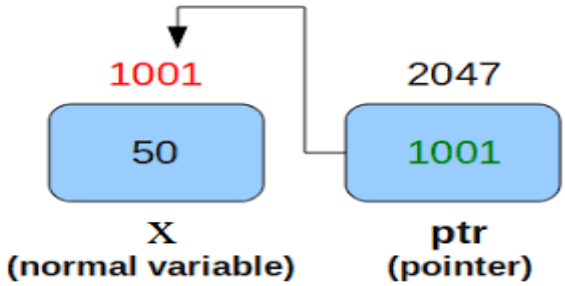
Here,
- **data_type** is the pointer's base type of C's variable types and indicates the type of the variable that the pointer points to.
- The asterisk (*: the same asterisk used for multiplication) which is indirection operator, declares a pointer.

Example:-

int *p;
float *ptr;
Note: - pointers can also declare in these ways

int* p;
int * p;

The graphical representation of pointers is shown in figure below. Here var is the variable and ptr is the pointer variable. ptr stores the address of variable x.



Initialize a Pointer

When we declare a pointer variable, it does not automatically point to any particular memory location. To initialize a pointer to point to a specific variable or memory location, we use the ampersand &(addressof) operator to get the address of that variable.

Syntax:-

Pointer_variable = &variable;

Example:-

```
int var = 10;
int * ptr;
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

Example

```
int *ptr = &var;
```

Dereference a Pointer

Once we have a pointer that points to a specific memory location, we can access or modify the value stored at that location by dereferencing the pointer. To dereference a pointer, we use the asterisk * symbol again, but this time in front of the pointer variable itself.

Example:-

```
int x = 42;
int *p = &x;
printf("%d\n", *p);
```

Program:-

```
#include<stdio.h>
main()
{
int x = 42;
int *p = &x;
printf("x=%d",*p);
}
```

Output:-

X=42

Swapping of two numbers using pointers

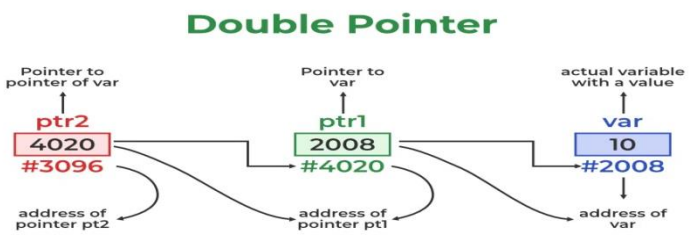
```
#include<stdio.h>
main()
{
int x = 42,y=33;
int *p = &x,*q=&y;
int temp;
printf("Before swapping ");
printf("\n x=%d",*p);
printf("\n y=%d",*q);
temp=*p;
*p=*q;
*q=temp;
printf("\n Afeter swapping ");
printf("\n x=%d",*p);
printf("\n y=%d",*q);
}
```

Output:-

```
Before swapping
x=42
y=33
Afeter swapping
x=33
y=42
```

Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



Declaration of Pointer to a Pointer in C

Declaring Pointer to Pointer is similar to declaring a pointer in C. The difference is we have to place an additional '*' before the name of the pointer.

Syntax:-

data_type_of_pointer **name_of_variable = & normal_pointer_variable;

Example:-

```
int a=40;
int *p,**q;
p=&a;
q=&p;
```

Program:-

```
#include<stdio.h>
main()
{
int a=10;
int i,n;
int *p,**q;
p=&a;
q=&p;
printf("a=%d",a);
printf("\n a=%d",*p);
printf("\n a=%d",**q);
}
```

Output:-

```
a=10
a=10
a=10
```

Pointer Arithmetic's

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations.

These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Comparison of pointers

1. Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

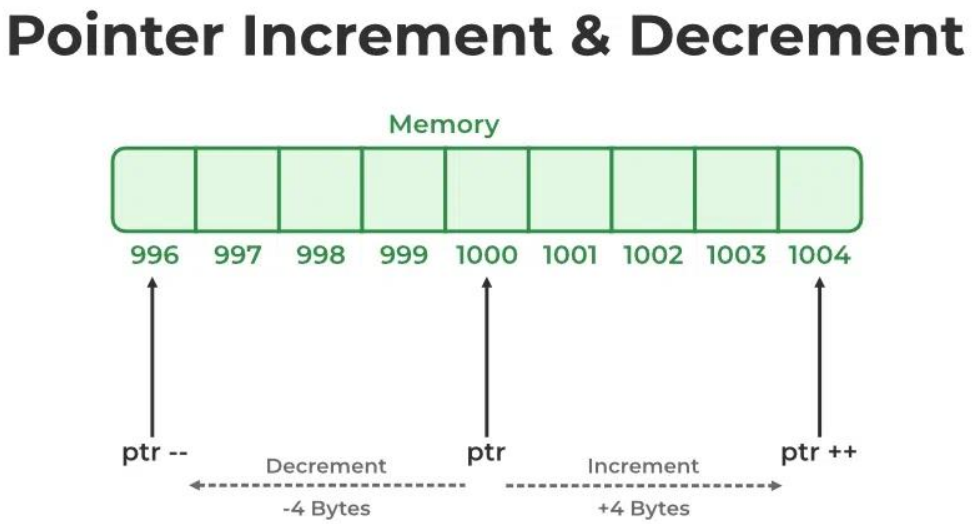
For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004**.

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996**.



Program:-

```
#include <stdio.h>
main()
{
   int a = 22;
   int *p = &a;
   printf("p = %u\n", p);
   p++;
   printf("p++ = %u\n", p);
   p--;
   printf("p-- = %u\n", p);

   float b = 22.22;
   float *q = &b;
   printf("q = %u\n", q    q++;
   printf("q++ = %u\n", q);
   q--;
   printf("q-- = %u\n", q);

   char c = 'a';
   char *r = &c;
   printf("r = %u\n", r);
   r++;
   printf("r++ = %u\n", r);
   r--;
   printf("r-- = %u\n", r);

}
```

Output:-

```
p = 1441900792
p++ = 1441900796
p-- = 1441900792
q = 1441900796
q++ = 1441900800
q-- = 1441900796
r = 1441900791
r++ = 1441900792
r-- = 1441900791
```

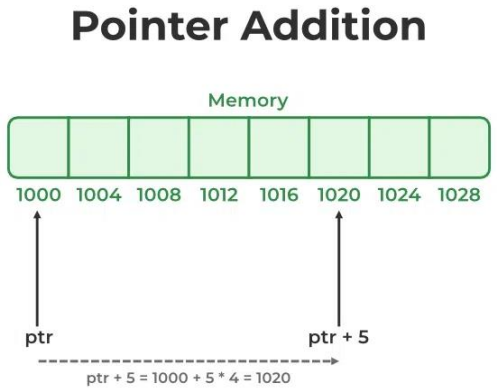
2. Addition of Integer to Pointer

When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

i.e pointer_value + size of datype * interger value

Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5,** then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) * 5 = 1020.**




```
#include <stdio.h>
main()
{
    int N = 4;
   int *ptr2;
   ptr2 = &N;
   printf("Pointer ptr2 before Addition: ");
   printf("%d \n", ptr2);
   ptr2 = ptr2 + 3;
  printf("Pointer ptr2 after Addition: ");
  printf("%d \n", ptr2);
}
```

Output:-

Pointer ptr2 before Addition: 6487572
Pointer ptr2 after Addition: 6487584

Program:- Addition of two pointers

```
#include <stdio.h>
main()
{
   int N = 4,M=6;
   int *ptr1,*ptr2;
   ptr1 = &N;
   ptr2 = &M;
   printf("Pointer ptr1 and ptr2 Addition: ");
   printf("%d \n",*ptr1+*ptr2);
  }
```

Output:-

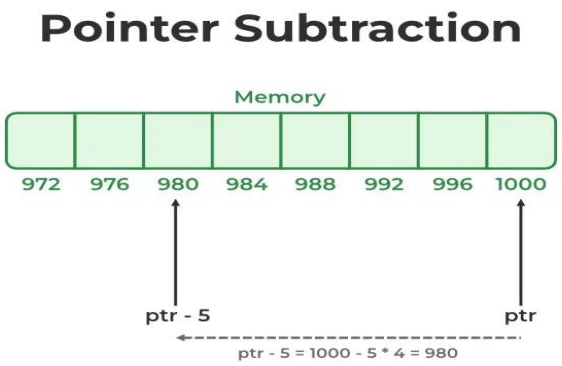
Pointer ptr1 and ptr2 Addition: 10

3. Subtraction of Integer to Pointer

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we subtract integer 5 from it using the expression, **ptr = ptr – 5,** then, the final address stored in the ptr will be **ptr = 1000 – sizeof(int) * 5 = 980.**




```
#include <stdio.h>
main()
{
   int N = 4;
  int *ptr2;
  ptr2 = &N;
```

```
   printf("Pointer ptr2 before Addition: ");
   printf("%d \n", ptr2);
   ptr2 = ptr2 - 3;
  printf("Pointer ptr2 after Subtraction: ");
  printf("%d \n", ptr2);
}
```

Output:-

Pointer ptr2 before Addition: 6487572
Pointer ptr2 after Addition: 6487566

Program:- subtraction of two pointers

```
#include <stdio.h>
main()
{
    int x = 6;
    int N = 4;
    int *ptr1, *ptr2;
    ptr1 = &x;
    ptr2 = &N;
    printf(" x = %d\n N=%d ", *ptr1, *ptr2);
    x = *ptr1 - *ptr2;
   printf("\n Subtraction of ptr1 and ptr2 is %d", x);

}
```

Output:-

x = 6
N=4
Subtraction of ptr1 and ptr2 is 2

4. Comparison of Pointers

We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C **>, >=, <, <=, = =, !=.** It returns true for the valid condition and returns false for the unsatisfied condition.

1. **Step 1:** Initialize the integer values and point these integer values to the pointer.
2. **Step 2:** Now, check the condition by using comparison or relational operators on pointer variables.
3. **Step 3:** Display the output.

```
#include <stdio.h>
main()
{
        // declaring array
        int a=20;

        // declaring pointer to array name
        int* ptr1 =&a;
        // declaring pointer to first element
        int* ptr2 = &a;

        if (ptr1 == ptr2)
         {
                printf("Both pointer values are equal ");
        }
        else
         {
                printf("Both pointer values are not equal.");
        }
}
```

Output:-

Both pointer values are equal

Accessing array elements using pointers

Single dimensional array:-

Array name is a constant pointer that points to the base address of the array [i.e. the first element of the array]. Elements of the array are stored in contiguous memory locations. They can be efficiently accessed by using pointers. Pointer variable can be assigned to an array. The address of each element is increased by one factor depending upon the type of data type. The factor depends on the type of pointer variable defined. If it is integer the factor is increased by 2 or 4.

Syntax:-

pointer_variable = array_name;

(**or)**

pointer_variable=&array_name;

Example:-

```
int x[5]={11,22,33,44,55}, *p;
p = x; (or) p=&x; (or) p = &x[0];
```

program:-

```
#include<stdio.h>
main()
{
int a[10];
int n,i,sum=0;
printf("Enter size of array:");
scanf("%d",&n);
printf("Enter elements:");
for(i=0;i<n;i++)
scanf("%d",(a+i));
printf("Array elements are :");
for(i=0;i<n;i++)
{
printf("\n %d",*(a+i));
}
}
```

Output:-

```
Enter size of array: 5
Enter elements: 10
11
12
13
14
Array elements are :
 10
 11
 12
 13
 14
```

Assigning 1-D Array to a Pointer Variable

program:-

```
#include<stdio.h>
main()
{
int a[10];
int n,i,sum=0;
int *p=a;
printf("Enter size of array:");
scanf("%d",&n);
printf("Enter elements:");
for(i=0;i<n;i++)
scanf("%d",(p+i));
printf("Sum of array elements are :");
for(i=0;i<n;i++)
{
        sum = sum+*(p+i);

}
printf("\n %d",sum);
}
```

Output:-

```
Enter size of array:5
Enter elements:1
2
3
4
5
Sum of array elements are: 15
```

Pointer with two dimensional arrays

In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element arr[i][j] of the array using the pointer expression ***(*(arr + i) + j)**.

Subscripting Pointer to an Array

Suppose *arr* is a 2-D array with 3 rows and 4 columns and *ptr* is a pointer to an array of 4 integers, and *ptr* contains the base address of array *arr*.

int arr[3][4];
int (*ptr)[4];
ptr = arr;

Since ptr is a pointer to the first row 2-D array i.e. array of 4 integers, ptr + i will point to i^{th} row. On dereferencing ptr + i, we get base address of i^{th} row. To access the address of j^{th} element of i^{th} row we can add j to the pointer expression *(ptr + i). So the pointer expression *(ptr + i) + j gives the address of j^{th} element of i^{th} row and the pointer expression *(*(ptr + i)+j) gives the value of the j^{th} element of i^{th} row.

We know that the pointer expression *(*(ptr + i) + j) is equivalent to subscript expression ptr[i][j]. So if we have a pointer variable containing the base address of 2-D array, then we can access the elements of array by double subscripting that pointer variable.

Program:-

```
#include<stdio.h>
main()
{
int a[10][10];
int n,m,i,j;
printf("Enter row and col size of array:");
scanf("%d%d",&n,&m);
printf("Enter elements:");
for(i=0;i<n; i++)
for(j=0;j<m; j++)
scanf("%d",(*(a+i)+j));
printf("Array elements are :\n");
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
printf("%d ",*(*(a+i)+j));
printf("\n");
}
}
```

Output:-

```
Enter row and col size of array:2
2
Enter elements:1
2
3
4
```

Array elements are :
1 2
3 4

Assigning 2-D Array to a Pointer Variable

Program:-

```
#include<stdio.h>
main()
{
int a[10][10];
int n,m,i,j,sum=0;
int (*p)[10];
p=a;
printf("Enter row and col size of array:");
scanf("%d%d",&n,&m);
printf("Enter elements:");
for(i=0;i<n;i++)
for(j=0;j<m;j++)
scanf("%d",(*(p+i)+j));
printf("Sum of array elements are :\n");
for(i=0;i<n;i++)
for(j=0;j<m;j++)
sum = sum+*(*(p+i)+j);
printf("%d ",sum);
}
```

Output:-

Enter row and col size of array: 2
2
Enter elements:1
2
3
4
Sum of array elements are :
10

Accessing String via a Pointer

When we point a char array to a pointer, we pass the base address of the array to the pointer. The pointer variable can be dereferenced using the asterisk * symbol in C to get the character stored in the address.

Example:-

char arr[] = "Hello";
char *ptr = arr; *// pointing pointer ptr to starting address of the array arr*

Program:-

```
#include<stdio.h>
    main()
 {
    char str[11] = "HelloWorld";
    char *ptr = str;
    printf("String is :”);
    while (*ptr != '\0')
   {
    printf("%c", *ptr);
    ptr++;
   }
}
```

Output:-

String is: HelloWorld

STRUCTURES

A structure is a group of variables known by one collective name. Within the structure, the individual variables (called members or fields) can be of different types.

Structure Type Declarations

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.

Syntax

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

Example:-

```
struct employee
{   int id;
   char name[20];
   float salary;
};
```

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, ...;
```

Example:-

```
struct employee
{
 int id;
   char name[50];
   float salary;
}e1,e2;
```

2. By struct keyword within main() function

// structure declared beforehand

struct *structure_name* ***variable1, variable2****, ...*;

Example:-

```
struct employee
{   int id;
   char name[50];
   float salary;
};
main()
{
struct employee e1, e2;
-------
-------
}
```

Access Members of a Structure

There are two types of operators used for accessing members of a structure.

1. By . (member or dot operator)
2. By -> (structure pointer operator)

1. By. (Member or dot operator)

The structure members can be accessed through the dot(.) or member operator.

Syntax:-

Structure_variable.structure_ member;

Example:-

```
struct employee
{   int id;
    char name[50];
    float salary;
}emp;
```

Now we can access structure members as

```
emp.id;
emp.name;
emp.salary;
```

Program:-

```
#include <stdio.h>
struct student {
    char name[50];
    int roll;
    float m1,m2,m3,m4,m5,tot;
} s;

int main()
{
    printf("Enter information:\n");
    printf("Enter name: ");
    scanf("%s",s.name);
    printf("Enter roll number: ");
    scanf("%d", &s.roll);
    printf("Enter marks1: ");
    scanf("%f", &s.m1);
    printf("Enter marks2: ");
    scanf("%f", &s.m2);
    printf("Enter marks3: ");
```

```
    scanf("%f", &s.m3);
    printf("Enter marks4: ");
    scanf("%f", &s.m4);
    printf("Enter marks5: ");
    scanf("%f", &s.m5);
    s.tot=s.m1+s.m2+s.m3+s.m4+s.m5;
    printf("Displaying Information:\n");
    printf("Name: ");
    printf("%s", s.name);
    printf("\nRoll number: %d", s.roll);
    printf("\nTotal Marks: %.1f\n", s.tot);
}
```

Output:-

```
Enter information:
Enter name: srikar
Enter roll number: 101
Enter marks1: 67
Enter marks2: 87
Enter marks3: 66
Enter marks4: 89
Enter marks5: 97
Displaying Information:
Name: srikar
Roll number: 101
Total Marks: 406.0
```

Initialize Structure Members

Initialization using Assignment Operator

You can initialize structure members individually using the assignment operator (=) after declaring the structure variable.

```
#include <stdio.h>
struct Point
{
    int x;
    int y;
};
```

```
main()
 {
   struct Point p1;
   p1.x = 5;
   p1.y = 10;
   printf("Coordinates of p1: (%d, %d)\n", p1.x, p1.y);
   }
```

Output:-

Coordinates of p1: (5, 10)

Initialization using Initializer List

You can also initialize structure members using an initializer list when declaring the structure variable.

```
#include <stdio.h>
struct Point
{
   int x;
   int y;
};

main()
{
   // Declare and initialize a structure variable using an initializer list
   struct Point p1 = {5, 10};

   printf("Coordinates of p1: (%d, %d)\n", p1.x, p1.y);

}
```

Output:-

Coordinates of p1: (5, 10)

Initialization using Designated Initializer List

Designated initializers allow you to specify the member you want to initialize explicitly. This is especially useful when you want to initialize only specific members of a structure.

```
#include <stdio.h>
struct Point {
   int x;
   int y;
};
 main()
 {
   // Declare and initialize a structure variable using designated initializers
   struct Point p1 = {.x = 5, .y = 10};

   printf("Coordinates of p1: (%d, %d)\n", p1.x, p1.y);
}
```

Output:-

Coordinates of p1: (5, 10)

Array of Structures

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Example:-

```
struct employee
{   int id;
   char name[50];
   float salary;
}emp[10];
```

Program:-

```
#include <stdio.h>
struct student {
   char name[50];
   int roll;
   float m1,m2,m3,m4,m5,tot;
};
```

```
 main()
{
        struct student s[5];
        int i,n;
        printf("Enter the no of students");
        scanf("%d",&n);
        printf("Enter student information:\n");
        for(i=0;i<n;i++)
        {
printf("student-%d information",i+1);
   printf("\nEnter name: ");
   scanf("%s",s[i].name);
   printf("Enter roll number: ");
   scanf("%d", &s[i].roll);
   printf("Enter marks1: ");
   scanf("%f", &s[i].m1);
   printf("Enter marks2: ");
   scanf("%f", &s[i].m2);
   printf("Enter marks3: ");
   scanf("%f", &s[i].m3);
   printf("Enter marks4: ");
   scanf("%f", &s[i].m4);
   printf("Enter marks5: ");
   scanf("%f", &s[i].m5);
   s[i].tot=s[i].m1+s[i].m2+s[i].m3+s[i].m4+s[i].m5;
}
 printf("Student Information:\n");
for(i=0;i<n;i++)
        {
   printf("Name: ");
   printf("%s", s[i].name);
   printf("\t\tRoll number: %d", s[i].roll);
   printf("\t\tTotal Marks: %.1f\n", s[i].tot);
   printf("\n");
}
}
```

Output:-

```
Enter the no of students2
Enter student information:
```

student-1 information
Enter name: srikar
Enter roll number: 101
Enter marks1: 66
Enter marks2: 77
Enter marks3: 88
Enter marks4: 99
Enter marks5: 87
student-2 information
Enter name: madhu
Enter roll number: 102
Enter marks1: 66
Enter marks2: 56
Enter marks3: 68
Enter marks4: 87
Enter marks5: 68
Student Information:
Name: srikar Roll number: 101 Total Marks: 417.0
Name: madhu Roll number: 102 Total Marks: 345.0

Nested structure

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.

Syntax:

```
struct name_1
{
   member1;
  member2;
  .
  .
  membern;

struct name_2
  {
     member_1;
     member_2;
     .
     .
    member_n;
  }, var1
} var2;
```

The member of a nested structure can be accessed using the following syntax:

Variable name of Outer_Structure.Variable name of Nested_Structure.data member to access;

Different ways of nesting structure

The structure can be nested in the following different ways:

a) By separate nested structure
b) By embedded nested structure.

a). By separate nested structure: In this method, the two structures are created, but the dependent structure should be used inside the main structure as a member.

Program:-

```
#include<stdio.h>
struct address
{
   char city[20];
   int pin;
   int phone;
};
struct employee
{
   char name[20];
   struct address add;
};
main ()
{
   struct employee emp;
   printf("Enter employee information\n");
   printf("Employee Name:");
   scanf("%s",emp.name);
   printf("Employee City:");
   scanf("%s",emp.add.city);
   printf("Employee Pin:");
   scanf("%d",&emp.add.pin);
   printf("Employee Phone:");
   scanf("%d",&emp.add.phone);
   printf("Printing the employee information....\n");
printf("name:%s\nCity:%s\nPincode:%d\nPhone:%d",emp.name,emp.add.city,emp.add.pin,emp.
add.phone);
}
```

Output:-

```
Enter employee information
Employee Name:srikar
Employee City:bhimavaram
Employee Pin:534201
Employee Phone:12345
Printing the employee information....
name: srikar
City: bhimavaram
Pincode: 534201
Phone: 12345
```

b) Embedded 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.

```
#include<stdio.h>
struct address
{
   char city[20];
   int pin;
   int phone;
         struct employee
         {
   char name[20];
} employee;
};

main ()
{
   struct address emp;
   printf("Enter employee information\n");
   printf("Employee Name:");
   scanf("%s",emp.employee.name);
   printf("Employee City:");
   scanf("%s",emp.city);
   printf("Employee Pin:");
   scanf("%d",&emp.pin);
   printf("Employee Phone:");
```

```
    scanf("%d",&emp.phone);
    printf("Printing the employee information....\n");

printf("name:%s\nCity:%s\nPincode:%d\nPhone:%d",emp.employee.name,emp.city,emp.pin,em
p.phone);
}
```

Output:-

```
Enter employee information
Employee Name:srikar
Employee City:bhimavaram
Employee Pin:534201
Employee Phone:12345
Printing the employee information....
name: srikar
City: bhimavaram
Pincode: 534201
Phone: 12345
```

Using typedef with structures

The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable .

Syntax

typedef <existing_name> <alias_name>

Example:-

typedef unsigned int unit;

Consider the below structure declaration:

```
typedef struct student
{
char name[20];
int age;
} stud;
stud s1,s2;
```

program:-

```
#include <stdio.h>
typedef struct student
{
char name[20];
int age;
}stud;
main()
{
stud s1;
printf("Enter the details of student s1: ");
printf("\nEnter the name of the student:");
scanf("%s",&s1.name);
printf("\nEnter the age of student:");
scanf("%d",&s1.age);
printf("\n Name of the student is : %s", s1.name);
printf("\n Age of the student is : %d", s1.age);
}
```

Output:-

```
Enter the details of student s1:
Enter the name of the student: Peter
Enter the age of student: 28
Name of the student is : Peter
Age of the student is : 28
```

2. By -> (structure pointer operator)

Pointer to structure holds the add of the entire structure.
The members of the structure can be accessed using a special operator called as an arrow operator (->).

Declaration

Following is the declaration for pointers to structures in C programming −

```
struct tagname *ptr;
```

For example − struct student *s

Accessing

It is explained below how to access the pointers to structures.

Ptr-> membername;

For example − s->sno, s->sname, s->marks;

Program:-

```
#include<stdio.h>
struct student{
  int sno;
  char sname[30];
  float marks;
};
main ( )
{
  struct student s;
  struct student *st;
  printf("enter sno, sname, marks:");
  scanf ("%d%s%f", & s.sno, s.sname, &s. marks);
  st = &s;
  printf ("details of the student are");
  printf ("Number = %d", st ->sno);
  printf ("name = %s", st->sname);
  printf ("marks =%f", st ->marks);
}
```

Output:-

Let us run the above program that will produce the following result −

```
enter sno, sname, marks:1 Lucky 98
details of the student are:
Number = 1
name = Lucky
marks =98.000000
```

UNIONS

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program.

Syntax:-

```
union [union tag]
{
  member definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

Example:-

```
union Data
 {
  int i;
  float f;
  char str[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. The memory occupied by a union will be large enough to hold the largest member of the union.

Program:-

```
#include <stdio.h>
#include <string.h>
 union Data {
  int i;
  float f;
```

```
  char str[20];
};
 main( )
{
  union Data data;
  printf( "Memory size occupied by data : %d\n", sizeof(data));
}
```

Output:-

Memory size occupied by data : 20

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access.

Program:-

```
#include <stdio.h>
#include <string.h>
 union Data {
  int i;
  float f;
  char str[20];
};
 main( )
{
  union Data data;
  data.i = 10;
  data.f = 220.5;
  strcpy( data.str, "C Programming");
  printf( "data.i : %d\n", data.i);
  printf( "data.f : %f\n", data.f);
  printf( "data.str : %s\n", data.str);
}
```

Output:-

data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming

Difference between Structure and Union in C

Parameter	Structure	Union
Keyword	A user can deploy the keyword **struct** to define a Structure.	A user can deploy the keyword **union** to define a Union.
Internal Implementation	The implementation of Structure in C occurs internally- because it contains separate memory locations allotted to every input member.	In the case of a Union, the memory allocation occurs for only one member with the largest size among all the input variables. It shares the same location among all these members/objects.
Accessing Members	A user can access individual members at a given time.	A user can access only one member at a given time.
Syntax	The Syntax of declaring a Structure in C is: struct [structure name] { type element_1; type element_2; . . } variable_1, variable_2, …;	The Syntax of declaring a Union in C is: union [union name] { type element_1; type element_2; . . } variable_1, variable_2, …;
Size	A Structure does not have a shared location for all of its members. It makes the size of a Structure to be greater than or equal to the sum of the size of its data members.	A Union does not have a separate location for every member in it. It makes its size equal to the size of the largest member among all the data members.
Value Altering	Altering the values of a single member does not affect the other members of a Structure.	When you alter the values of a single member, it affects the values of other members.

Storage of Value	In the case of a Structure, there is a specific memory location for every input data member. Thus, it can store multiple values of the various members.	In the case of a Union, there is an allocation of only one shared memory for all the input data members. Thus, it stores one value at a time for all of its members.
Initialization	In the case of a Structure, a user can initialize multiple members at the same time.	In the case of a Union, a user can only initiate the first member at a time.

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 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.

Need of Bit Fields in C

- Reduces memory consumption.
- To make our program more efficient and flexible.
- Easy to Implement.

Declaration of C Bit Fields

Bit-fields are variables that are defined using a predefined width or size.

Syntax

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

Example:-

```
struct dob
{
   unsigned int date: 5;
   unsigned int month: 4;
   unsigned int year: 12;
};
```

Program:-

```
#include <stdio.h>
struct time
{
        unsigned int hours:5;
        unsigned int minutes:6;
        unsigned int seconds:6;

};
main()
 {
 struct time t={10,30,45};
 printf("Displayy time as:");
 printf("%d:%d:%d",t.hours,t.minutes,t.seconds);
 }
```

Output:-

Display time as:10:30:45