C PROGRAMMING AND DATA STRUCTURES

UNIT-II

TOPICS:-

1. Functions
2. types of functions
3. Recursion and argument passing
4. Pointers
5. storage allocation
6. pointers to functions
7. expressions involving pointers
8. Storage classes – auto, register, static, extern
9. Structures
10. Unions
11. Strings
12. string handling functions
13. Command line arguments.

Topic 1: Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is main( ), and programmer can define more than one function in a program.
A function can also be referred as a method or a sub-routine or a procedure, etc.

Topic 2: Types of Functions

There are two types of functions in C programming.

1. **Library Functions**: are the inbuilt or pre definedfunctions which are declared in the C header files. Ex:- scanf( ), printf( ), sqrt( ),pow( ),gets( ), puts( ), ceil( ), floor( ) etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations.
For example, **printf** is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as **printf/scanf** we need to include **stdio.h** in our program which is a header file that contains all the library functions regarding standard input/output.

Types of user defined functions:-A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different methods to define function.

- function without arguments and without return value
- function with arguments and with return value
- function without arguments and with return value
- function with arguments and without return value

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Example without return value:

```
void hello( )
{
printf("hello c");
}
```

Example with return value:

```
int get( )
{
return 10;
}
```

Argument:-An argument is referred to the values that are passed from one function to another function. C functions exchange information by means of parameters or arguments.

Function Aspects:

There are three aspects of a C function.

Function declaration : A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

Function call : Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

Function definition: It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

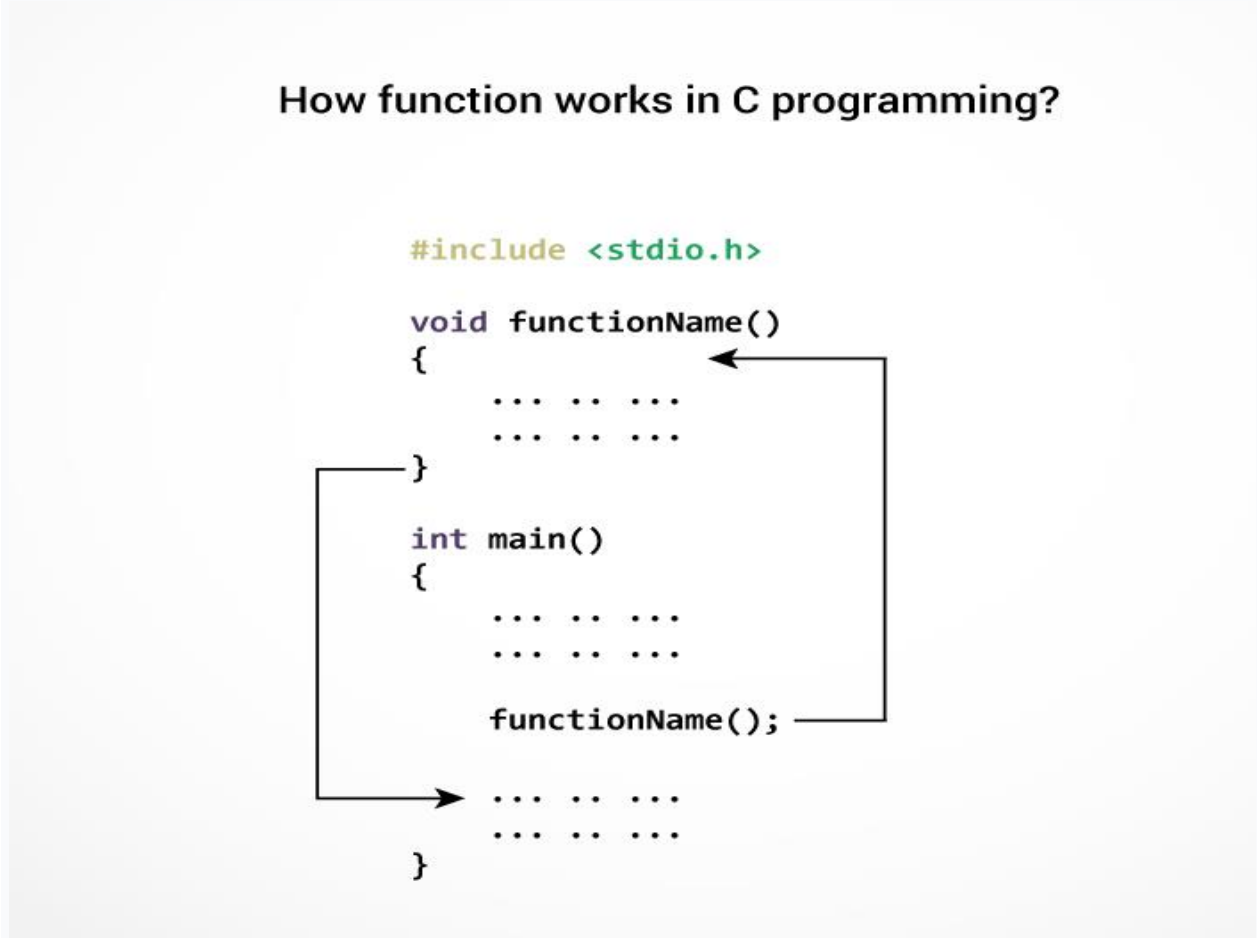
SN	C function aspects	Syntax
1	Function declaration	return_type function_name(argument list);
2	Function call	function_name(argument_list);

3	Function definition	return_type function_name (argument list)
		{
		function body;
		}

The syntax of creating function in c language is given below:

```
return_type  function_name(data_type  parameter...)
{
//code to be executed
}
```

How user-defined function works?



The execution of a C program begins from the main( ) function.

When the compiler encounters functionName( );, control of the program jumps to

```
voidfunctionName( )
```

And, the compiler starts executing the codes inside functionName( ).

The control of the program jumps back to the main( ) function once code inside the function definition is executed.

Example program for function with argument and with return value

```
#include<stdio.h>
int sum(int, int);  /*prototype*/
void main( )
{
   int a,b,result;
  printf("\nGoing to calculate the sum of two numbers:");
  printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);  /*function call*/
   printf("\nThe sum is : %d",result);
}
int sum(int a, int b)  /*function definition*/
{
   return a+b;
}
```

Input and Output:-

```
Going to calculate the sum of two numbers:
Enter two numbers:10
20
The sum is : 30
```

Example program for function with argument and without return value

```
#include<stdio.h>
void sum(int, int);  /*prototype*/
void main( )
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   sum(a,b);  /*function call*/
}
void sum(int a, int b)  /*function definition*/
{
  printf("\nThe sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example program for function without argument and with return value

```
#include<stdio.h>
int sum(void);  /*prototype*/
void main( )
{
   int result;
   printf("\nGoing to calculate the sum of two numbers:");
   result = sum( );  /*function call*/
   printf("%d",result);
}
```

```
int sum( )  /*function definition*/
{
  int a,b;
   printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example program for function without arguments and without return value

```
#include<stdio.h>
void sum(void);  /*prototype*/
void main( )
{
   printf("\nGoing to calculate the sum of two numbers:");
   sum();  /*function call*/
}
void sum(void)  /*function definition*/
{
   int a,b;
  printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   printf("The sum is %d",a+b);
}
```

Output

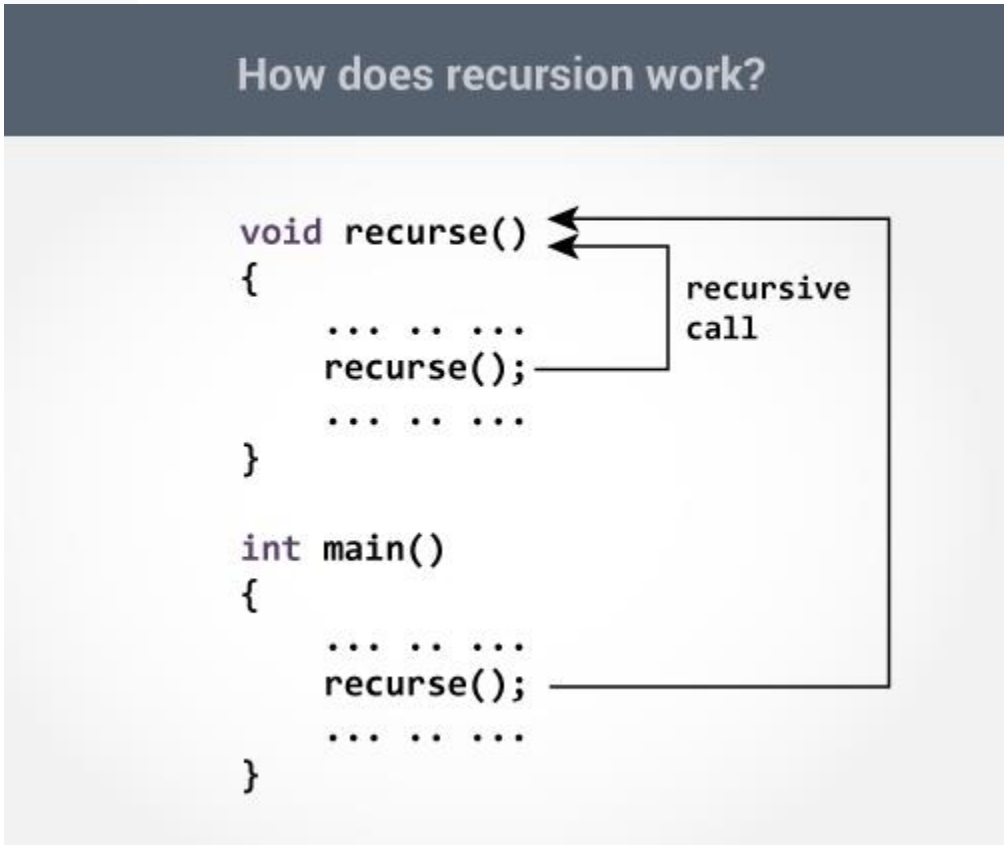
Going to calculate the sum of two numbers:
Enter two numbers 10
24
The sum is 34

---*---**

Topic :3 Recursion and argument passing

3.1 Recursion:-

A function that calls itself is known as a recursive function. And, this technique is known as recursion.



The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Example program

A simple example of recursion would be:

```
void recurse( )
{
   recurse( ); /* Function calls itself */
}

int  main( )
{
   recurse( ); /* Sets off the recursion */
   return 0;
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash.

Example 1 Program for recursion

```
#include <stdio.h>
void printnum(int begin)
{
   printf( "%d\t", begin);
   if(begin < 9)
   {
      printnum (begin + 1);
   }
   printf("%d", begin);
}
int main( )
{
   printnum(0);
}
```

Output:

0 1 2 3 4 5 6 7 8 9

Example 2 Program for sum of n natural numbers using recursion

```
#include <stdio.h>
int sum(int n);
int main( )
{
   int number, result;
   printf("Enter a positive integer: ");
   scanf("%d", &number);
   result = sum(number);
   printf("sum = %d", result);
   return 0;
}
int sum(int n)
{
   if (n != 0)
      /*sum( ) function calls itself*/
      return n + sum(n-1);
   else
      return n;
}
```

Output

Enter a positive integer:3
sum = 6

3.2 Argument passing (parameter passing mechanisms)

Functions can be invoked(called) in two ways: **Call by Value** or **Call by Reference**. The parameters passed to function are called *actual parameters* whereas the parameters received by function are called *formal parameters*. (or) The parameters in function call are called actual parameters, whereas the parameters in function definition are called formal parameters.

Call by Value

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function. Let's take a look at a call by value example:

```
#include <stdio.h>
void call_by_value(int x)
{
        printf("Inside call_by_value x = %d before adding 10.\n", x);
        x += 10;
        printf("Inside call_by_value x = %d after adding 10.\n", x);
}

main( )
{
        int a=10;

        printf("a = %d before function call_by_value.\n", a);
        call_by_value(a);
        printf("a = %d after function call_by_value.\n", a);
}
```

Output:-

```
a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.
Inside call_by_value x = 20 after adding 10.
a = 10 after function call_by_value.
```

Call by Reference

If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example:

```
#include <stdio.h>
void call_by_reference(int *y)
{
        printf("Inside call_by_reference y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}
main( )
 {
        int b=10;
        printf("b = %d before function call_by_reference.\n", b);
        call_by_reference(&b);
        printf("b = %d after function call_by_reference.\n", b);

}
```

Output:-
b = 10 before function call_by_reference.
Inside call_by_reference y = 10 before adding 10.
Inside call_by_reference y = 20 after adding 10.
b = 20 after function call_by_reference.

---***---

Topic 4: Pointers

Definition:- Pointers (pointer variables) are special variables that are used to store addresses rather than values. Pointer is a variable that stores address of another variable.

Like variables, pointers in C programming have to be declared before they can be used in the program.

Declaration of pointer:-

Syntax for declaring pointer:-

data_type *pointer_variable_name;

Ex:-

```
int *p1;
int * p2;
```

Initialization of pointer:-

```
Ex:-
int *pc, c;
c = 5;
pc = &c;
```

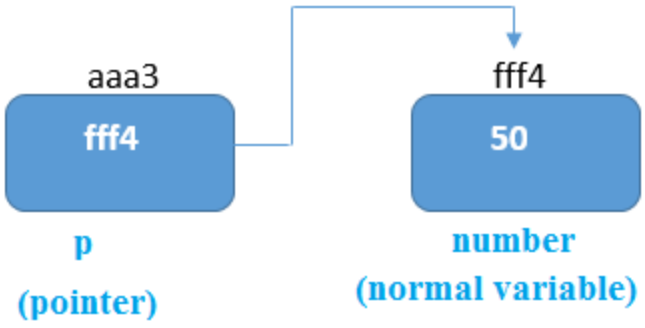
Here, 5 is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

Get Value of variable Pointed by Pointers:-

To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);   // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.



Example Program

```
#include<stdio.h>
main()
{
int number=50;
int *p;
p=&number;
printf("Address of p variable is %x \n",p);
printf("Value of p variable is %d \n",*p);
}
```

Output

Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50

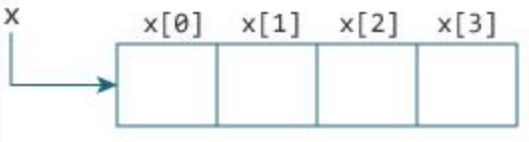
4.1 pointer and arrays

Pointer and 1d-array:-

Pointer can also be defined to an array or 1d-array.

Ex:- int a[4]={4,7,9,2};
int *p;
p=&a[0]; or p=a;

Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.



From the above example, it is clear that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example Program

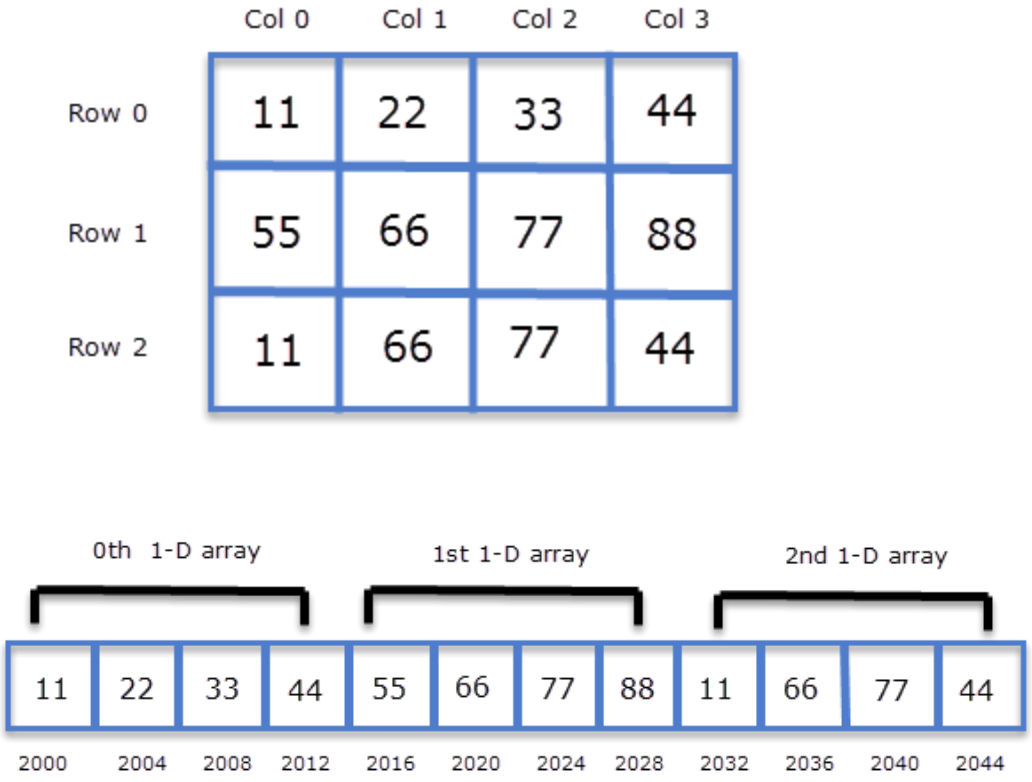
```
#include<stdio.h>
main()
{
  int x[4] = {1, 2, 3,9};
  int *p = x;
  int I;
  for ( i = 0; i < 3; i++)
  {
    printf("%d", *(x+i));
  }
}
```

Output:
1 2 3

Pointer and 2d-array:-

Pointer can also be defined to an array or 1d-array.

Ex:- int a[2][2]={{4,7},{9,2}};
int *p;
p=&a[0][0]; or p=a;

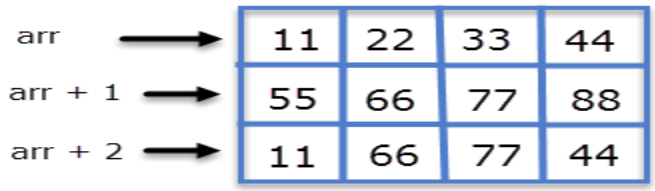


Note:- name of a 1-D array is a constant pointer to the 0th element. In the case, of a 2-D array, 0th element is a 1-D array. Hence in the above example, the type or base type of arr is a pointer to an array of 4 integers. Since pointer arithmetic is performed relative to the base size of the pointer. In the case of arr, if arr points to address 2000 then arr + 1 points to address 2016 (i.e 2000 + 4*4).

In the case of a 2-D array, 0th element is a 1-D array. So the name of the array in case of a 2-D array represents a pointer to the 0th 1-D array. Therefore in this case arr is a pointer to an array of 4 elements. If the address of the 0th 1-D is 2000, then according to pointer arithmetic (arr + 1) will represent the address 2016, similarly (arr + 2) will represent the address 2032.

From the above discussion, we can conclude that:

arr points to 0th 1-D array.
(arr + 1) points to 1st 1-D array.
(arr + 2) points to 2nd 1-D array.



*(arr + i) points to the address of the 0th element of the 1-D array. So,
*(arr + i) + 1 points to the address of the 1st element of the 1-D array
*(arr + i) + 2 points to the address of the 2nd element of the 1-D array

Hence we can conclude that:
*(arr + i) + j points to the base address of jth element of ith 1-D array.

On dereferencing *(arr + i) + j we will get the value of jth element of ith 1-D array.

*(*(arr + i) + j)

Example Program:-

```
#include<stdio.h>
imain( )
{
   int arr[3][4] = {
                    {11,22,33,44},
                    {55,66,77,88},
                    {11,66,77,44}
                  };
   int i, j;
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 4; j++)
      {
         printf("%d\t",*( *(arr + i) + j) );
      }
      printf("\n");
   }
}
```

Output:-

```
11      22      33      44
55      66      77      88
11      66      77      44
```

---***---

Topic 5: Storage allocation (dynamic memory allocation)

Memory allocation is the process of setting memory in a program to be used to store variables, arrays, structures, etc.
There are two basic types of memory allocation:

The process of allocating memory at compile time is known as static memory allocation.

Ex:- int a,b;

For variables a and b 2 bytes of memory is allocated at compile time itself – static allocation

The process of allocating memory at runtime is known as dynamic memory allocation.

Dynamic memory allocation is done with help of pointers.
The predefined functions which are used to allocate memory at runtime are called dynamic memory allocation functions.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

malloc( ):- This function is used to allocate memory at run time for a variable.

Syntax of malloc( ):

```
ptr = (castType*) malloc(size);
```

Example:

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of `float` is 4 bytes.

calloc( ):- This function is used to allocate memory at run time for an array.

Syntax of calloc( ):

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type `float`.

free( ):- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free( ):

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

realloc( ):- If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc( ):-

```
ptr = realloc(ptr, x);
```

Here, ptr is reallocated with a new size x.

Example program for malloc( ) and free( )

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *n;
   n=(int *)malloc(4);
   printf("Enter n value: ");
   scanf("%d", n);
   printf("value of n= %d",*n);
   free(n);
   return 0;
}
```

Output:-

```
Enter n value: 56
value of n= 56
```

Example program for calloc( ) and free( )

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
   int n, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   ptr = (int *) calloc(n,sizeof(int));
   printf("Enter elements: ");
   for(i=0;i<n;i++)
   {
      scanf("%d",(ptr + i));
   }
   printf("elements of array are:\n");
   for(i=0;i<n;++i)
   {
      printf("%d\t",*(ptr + i));
   }
   free(ptr);
   return 0;
}
```

Output:-

```
Enter number of elements: 5
Enter elements: 5
8
4
9
12
elements of array are:
5        8        4        9        12
```

Example program for realloc( ) and free( )

```
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
main( )
{
   char *a;
   a=(char *)malloc(10);
   strcpy(a,"hyderabad");
   printf("string is %s",a);
   a=realloc(a,15);
   strcpy(a,"secunderabad");
   printf("string now is %s",a);
   free(a);
}
```

Output:-

string is hyderabad

string now is secunderabad

---*---

Topic 6: Pointer to a function

It is possible to declare a pointer pointing to a function and also function can be called using pointer.

Syntax:-

function_return_type(*Pointer_name)(function argument list)

Example:- int (*f2p) (int, int);

Example Program:-

```
int sum (int num1, int num2)
{
   return num1+num2;
}
main( )
{

   int (*f2p) (int, int);
   f2p = sum;
```

```
    //Calling function using function pointer
    int op1 = f2p(10, 13);

    //Calling function in normal way using function name
    int op2 = sum(10, 13);
    printf("Output1: Call using function pointer: %d",op1);
    printf("\nOutput2: Call using function name: %d", op2);
}
```

Output:

Output1: Call using function pointer: 23
Output2: Call using function name: 23

Note:-

Pointer operators:-

Operator	Meaning
*	Serves 2 purpose 1. Declaration of a pointer 2. Returns the value of the referenced variable
&	Serves only 1 purpose • Returns the address of a variable

---***---

Topic 7: Pointer Expressions and arithmetic

Like other variables pointer variables can be used in expressions.

Pointer Assignments

You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward. For example:

```
int s = 56;
int *ptr1, *ptr2;
ptr1 = &s;
ptr2 = ptr1;
```

Program

```
main( )
{
int s = 56;
int *ptr1, *ptr2;
ptr1 = &s;
ptr2 = ptr1;
/* print the value of s twice */
printf("Values at ptr1 and ptr2: %d %d \n", *ptr1, *ptr2);
/* print the address of s twice */
printf("Addresses pointed to by ptr1 and ptr2: %p %p",ptr1,ptr2);
}
```

Output

Values at ptr1 and ptr2: 56 56
Addresses pointed to by ptr1 and ptr2: 0240FF20 0240FF20

Pointer Arithmetic

We can perform addition and subtraction of integer constant from pointer variable.

Addition

```
ptr1 = ptr1 + 2;
```

subtraction

```
ptr1 = ptr1 - 2;
```

We cannot perform addition, multiplication and division operations on two pointer variables.

For Example:

ptr1 + ptr2 is not valid

However we can subtract one pointer variable from another pointer variable. We can use increment and decrement operator along with pointer variable to increment or decrement the address contained in pointer variable.

For Example:

```
ptr1++;
ptr2--;
```

Multiplication

Example:

```
int x = 10, y = 20, z;
int *ptr1 = &x;
int *ptr2 = &y;
z = *ptr1 * *ptr2 ;
```

Will assign 200 to variable z.

Division

there is a blank space between '/' and * because the symbol /*is considered as beginning of the comment and therefore the statement fails.

```
Z=5*-*Ptr2/ *Ptr1;
```

If Ptr1 and Ptr2 are properly declared and initialized pointers, then the following statements are valid:

```
Y=*Ptr1**Ptr2;
Sum=sum+*Ptr1;
*Ptr2=*Ptr2+10;
*Ptr1=*Ptr1+*Ptr2;
*Ptr1=*Ptr2-*Ptr1;
```

```
Ptr1 = Ptr1 + 1 = 1000 + 2    = 1002;
Ptr1 = Ptr1 + 2 = 1000+ (2*2) = 1004;
Ptr1 = Ptr1 + 4 = 1000+ (2*4) = 1008;
Ptr2 = Ptr2 + 2 = 3000+ (2*2) = 3004;
Ptr2 = Ptr2 + 6 = 3000+ (2*6) = 3012;
```

Here addition means bytes that pointer data type hold are subtracted number of times that is subtracted to the pointer variable.

If Ptr1 and Ptr2 are properly declared and initialized pointers then, 'C' allows adding integers to a pointer variable.

EX:

```
int a=5, b=10;
int *Ptr1,*Ptr2;
Ptr1=&a;
Ptr2=&b
```

If Ptr1 & Ptr2 are properly declared and initialized, pointers then 'C' allows to subtract integers from pointers. From the above example,

```
Ptr1 = Ptr1 – 1 = 1000-2  = 998;
Ptr1 = Ptr1 - 2 = 1000-4  = 996;
Ptr1 = Ptr1 - 4 = 1000-8  = 992;
Ptr2 = Ptr2 - 2 = 3000-4  = 2996;
Ptr2 = Ptr2 - 6 = 3000-12 = 2988;
```

Here the subtraction means byte that pointer data type hold are subtracted number of times that is subtracted to the pointer variable.

If Ptr1 & Ptr2 are properly declared and initialize pointers, and both points to the elements of the same type. "Subtraction of one pointer from another pointer is also possible".

NOTE: this operation is done when both pointer variable points to the elements of the same array.

EX:

P2- P1 (It gives the number of elements between p1 and p2)

Pointer Increment and Scale Factor

We can use increment operator to increment the address of the pointer variable so that it points to next memory location.
The value by which the address of the pointer variable will increment is not fixed. It depends upon the data type of the pointer variable.

For Example:

```
int *ptr;
ptr++;
```

It will increment the address of pointer variable by 2. So if the address of pointer variable is 2000 then after increment it becomes 2002.
Thus the value by which address of the pointer variable increments is known as scale factor. The scale factor is different for different data types as shown below:

Char	1 Byte
Int	2 Byte
Short int	2 Byte
Long int	4 Byte
Float	4 Byte
Double	8 Byte
Long double	10 Byte

---*---

Topic 8: Storage Classes

Every variable in C programming has two properties: type and storage class.
Type refers to the data type of a variable.
Each variable has a storage class which defines the features of that variable. It tells the compiler about where to store the variable, its initial value, scope (visibility level) and lifetime (global or local).
Each variable has a storage class which define the following things:
Scopei.e where the value of the variable would be available inside a program.
default initial value i.e if we do not explicitly initialize that variable, what will be its default initial value.
lifetime of that variable i.e for how long will that variable exist.
Thus a storage class is used to represent the information about a variable.
There are 4 types of storage class:-
automatic
external
static
register

automatic storage class:-

The default storage class of all local variables (variables declared inside block or function) is auto storage class. Variable of auto storage class has the following properties...

Property	Description
Keyword	auto
Storage	Computer Memory (RAM)
Default Value	Garbage Value
Scope	Local to the block in which the variable is defined
Life time	Till the control remains within the block in which variable is defined

Example Program 1

```
#include<stdio.h>
#include<conio.h>
main( )
{
 auto  int a=10;
   {
      auto int a=20;
       printf("%d",a);
   }
   printf("\n%d",a);
}
```

Output:-

```
20
10
```

External storage class:-

The default storage class of all global variables (variables declared outside function) is external storage class. Variable of external storage class has the following properties...

Property	Description
Keyword	extern
Storage	Computer Memory (RAM)
Default Value	Zero
Scope	Global to the program (i.e., Throughout the program)
Life time	As long as the program's execution does not comes to end

Example Program 1

```
int n=10;
int main( )
{
printf("%d\t",n);       ->  Output: 10
n=n+10;
function1( );
printf("%d\t",n);       ->  Output: 30
}

int function1( )
{
printf("%d",n);         -> Output: 20
n=n+ 10;
}
```

Output:-

10 20 30

Static storage class:-

Variable of static storage class has the following properties...

Property	Description
Keyword	static
Storage	Computer Memory (RAM)
Default Value	Zero
Scope	Local to the block in which the variable is defined
Life time	The value of the persists between different function calls (i.e., Initialization is done only once)

Example Program 1

```
void function1( )
{
static int n=0;
n++;
printf("%d1n",n);
}
int main( )
{
int i;
for(i=0;i<3;i++)
```

```
funtion1( );
}
```
Output:-
1
2
3

Register storage class:-

The register variables enable faster accessibility compared to other storage class variables. As the number of registers inside the CPU is very less we can use very less number of register variables. Variable of register storage class has the following properties...

Property	Description
Keyword	register
Storage	CPU Register
Default Value	Garbage Value
Scope	Local to the block in which the variable is defined
Life time	Till the control remains within the block in which variable is defined

Example Program 1

```
#include<stdio.h>
main( )
{
   register int a=15;
   printf("%d", a);
}
```
Output:-
15

Storage Class	Keyword	Memory Location	Default Value	Scope	Life Time
Automatic	auto	Computer Memory (RAM)	Garbage Value	Local to the block in which the variable has defined	Within function
External	extern	Computer Memory (RAM)	Zero	Global to the program (i.e., Throughout the program)	Till the end of the main program Maybe declared anywhere in the program
Static	static	Computer Memory (RAM)	Zero	Local to the block in which the variable has defined	Till the end of the main program, Retains value between multiple functions call
Register	register	CPU Register	Garbage Value	Local to the block in which the variable has defined	Within the function

Topic 9: Structures

INTRODUCTION TO STRUCTURES

consider following example:
An entity Student may have its name (string), roll number (int), marks (float).
To store such type of information regarding an entity student, we have the following approaches:

- Construct individual arrays for storing names, roll numbers, and marks.
- Use a special data structure to store the collection of different data types.

Definition:- Structure is collection of elements of different datatypes.

Declaration of structure:-

The struct keyword is used to define the structure.

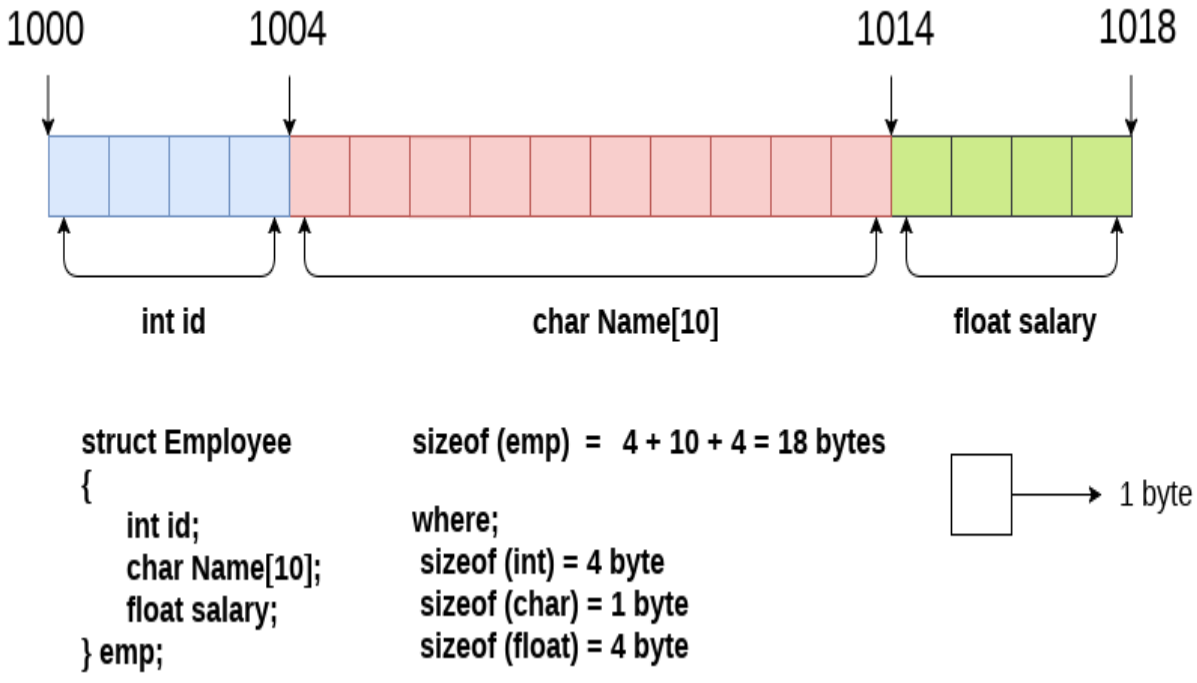
Syntax:-

```
struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

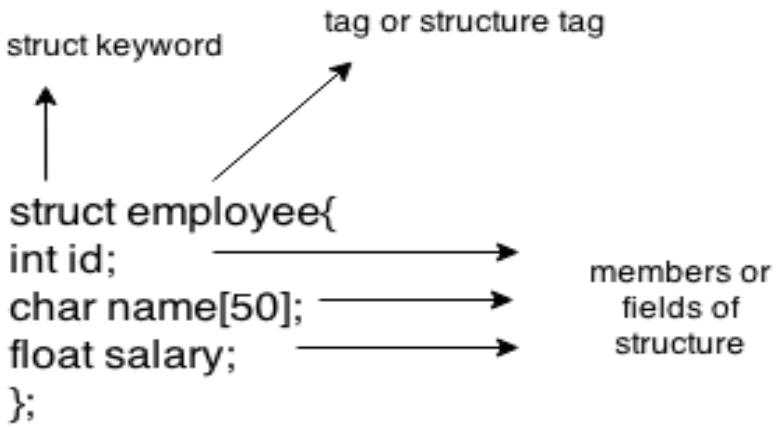
Let's see the example to define a structure for an entity employee in c.

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

The following image shows the memory allocation of the structure employee that is defined in the above example.



Here, struct is the keyword; employee is the name of the structure; id, name, and salary are the members or fields of the structure. Let's understand it by the diagram given below:



Declaring structure variable:-

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

By struct keyword within main( ) function

By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

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

Now write given code inside the main( ) function.

```
struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

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

Accessing members of the structure:-

There are two ways to access structure members:

By . (member or dot operator)

By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

```
p1.id
```

Initialization of structures:-
It can be done in 2 ways.

1) At the time of compilation
2) At the time of execution

At the time of compilation:-

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
   float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
main( )
{
   //store first employee information
   e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   e1.salary=56000;
   //store second employee information
   e2.id=102;
   strcpy(e2.name, "James Bond");
   e2.salary=126000;
    //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
   printf( "employee 1 salary : %f\n", e1.salary);
   //printing second employee information
   printf( "employee 2 id : %d\n", e2.id);
   printf( "employee 2 name : %s\n", e2.name);
   printf( "employee 2 salary : %f\n", e2.salary);
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

At the time of execution:-

```
#include <stdio.h>
#include <string.h>
struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};
 main( )
{
  struct Books b1;       /* Declare Book1 of type Book */
  struct Books b2;       /* Declare Book2 of type Book */
  printf("enter book1 details:\n");
  gets(b1.title);
  gets(b1.author);
  gets(b1.subject);
  scanf("%d",&b1.book_id);
   printf("enter book1 details:\n");
  gets(b2.title);
  gets(b2.author);
  gets(b2.subject);
  scanf("%d",&b2.book_id);
   /* print Book1 info */
  printf( "Book 1 title : %s\n", b1.title);
  printf( "Book 1 author : %s\n", b1.author);
  printf( "Book 1 subject : %s\n", b1.subject);
  printf( "Book 1 book_id : %d\n", b1.book_id);
  /* print Book2 info */
  printf( "Book 2 title : %s\n", b2.title);
  printf( "Book 2 author : %s\n", b2.author);
  printf( "Book 2 subject : %s\n", b2.subject);
  printf( "Book 2 book_id : %d\n", b2.book_id);
}
```

ARRAY OF STRUCTURES

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
struct student
{
   char name[20];
   int id;
   float marks;
};
struct student s1,s2,s3;
```

However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. However, c enables us to declare an array of structures.

An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures is also known as the collection of structures.

Syntax:- struct tagname arrayname[size];
Ex:- struct student s[20];

Example Program:-

```
#include<stdio.h>
#include <string.h>
struct student
{
int rollno;
char name[10];
};
int main( )
{
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++)
{
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
```

```
printf("\nStudent Information List:");
for(i=0;i<5;i++)
{
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

Input & Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

NESTED STRUCTURES

Structure within structure is known as nested structure.
The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
```

```
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

2) 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. Consider the following example.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
    int dd;
    int mm;
    int yyyy;
   }doj;
}emp1;
```

Accessing Nested Structure:-

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

```
e1.doj.dd
e1.doj.mm
e1.doj.yyyy
```

Example Program:-

```
#include <stdio.h>
#include <string.h>
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
    int dd;
    int mm;
    int yyyy;
   }doj;
}e1;
```

```
int main( )
{
  //storing employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.doj.dd=10;
  e1.doj.mm=11;
  e1.doj.yyyy=2014;

  //printing first employee information
  printf( "employee id : %d\n", e1.id);
  printf( "employee name : %s\n", e1.name);
  printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e
1.doj.yyyy);
  return 0;
}
```

Output:
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014

PASSING STRUCTURE TO FUNCTION IN C

It can be done in below 3 ways.

1. Passing structure to a function by value
2. Passing structure to a function by address(reference)
3. No need to pass a structure – Declare structure variable as global

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY VALUE:

In this program, the whole structure is passed to another function by value.

```
#include <stdio.h>
#include <string.h>
 struct student
{
        int id;
        char name[20];
        float percentage;
};
 void func(struct student record);
```

```
 int main( )
{
        struct student record;

        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
         func(record);
        return 0;
}

void func(struct student record)
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function.

```
#include <stdio.h>
#include <string.h>
 struct student
{
        int id;
        char name[20];
        float percentage;
};
void func(struct student *record);
 int main( )
{
       struct student record;
        record.id=1;
       strcpy(record.name, "Raju");
       record.percentage = 86.5;
        func(&record);
       return 0;
}
```

```
 void func(struct student *record)
{
        printf(" Id is: %d \n", record->id);
        printf(" Name is: %s \n", record->name);
        printf(" Percentage is: %f \n", record->percentage);
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program.

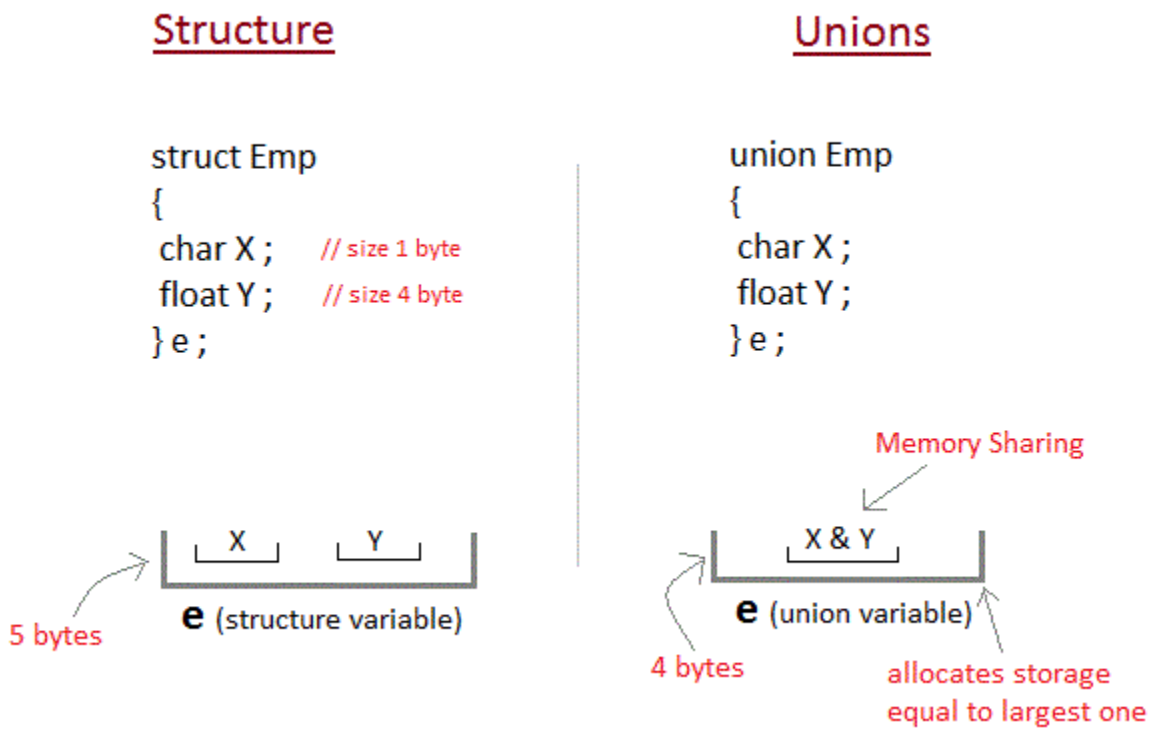
```
#include <stdio.h>
#include <string.h>
 struct student
{
        int id;
        char name[20];
        float percentage;
};
struct student record; // Global declaration of structure
 void structure_demo( );
 int main( )
{
        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
         structure_demo( );
        return 0;
}

void structure_demo( )
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

TOPIC 10: UNION

Unions are conceptually similar to structures. Union is collection of elements of different datatypes. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage. In structure each member has its own storage location, whereas all members of union uses a single shared memory location which is equal to the size of its largest data member.



A union is declared using the union keyword.

Structure vs union

Difference Between Structure and Union :

Structure	Union
i. Access Members	
We can access all the members of structure at anytime.	Only one member of union can be accessed at anytime.
ii. Memory Allocation	
Memory is allocated for all variables.	Allocates memory for variable which variable require more memory.
iii. Initialization	
All members of structure can be initialized	Only the first member of a union can be initialized.
iv. Keyword	
'struct' keyword is used to declare structure.	'union' keyword is used to declare union.
v. Syntax	

`struct struct_name`
`{`
`  structure element 1;`
`  structure element 2;`
`  ----------`
`  ----------`
`  structure element n;`
`}struct_var_nm;`
	`union union_name`
	`{`
	`  union element 1;`
	`  union element 2;`
	`  ----------`
	`  ----------`
	`  union element n;`
	`}union_var_nm;`

vi. Example	

`struct item_mst`
`{`
`  int rno;`
`  char nm[50];`
`}it;`
	`union item_mst`
	`{`
	`  int rno;`
	`  char nm[50];`
	`}it;`

TOPIC 11: STRINGS

string can be defined as the one-dimensional array of characters terminated by a null character('\0').
Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends.

Declaration and Initialization of string:-

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array in C language.

1. **char** ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

As we know, array index starts from 0, so it will be represented as in the figure given below.

While declaring string, size is not mandatory. So we can write the above code as given below:

1. **char** ch[ ]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

We can also define the string by the string literal in C language. For example:

1. **char** ch[ ]="javatpoint";

In such case, '\0' will be appended at the end of the string by the compiler.

String Input: Read a String

When writing interactive programs which ask the user for input, C provides the scanf( )

When we use scanf( ) to read, we use the "%s" format specifier without using the "&" to access the variable address because an array name acts as a pointer. For example:

```
#include <stdio.h>
int main( )
{
char name[10];
int age;
printf("Enter your first name and age: \n");
scanf("%s %d", name, &age);
printf("You entered: %s %d",name,age);
}
```

Input and Output:

```
Enter your first name and age:
John_Smith 48
```

The problem with the scanf function is that it never reads entire Strings in C. It will halt the reading process as soon as whitespace, form feed, vertical tab, newline or a carriage return occurs.

Suppose we give input as "GIST College" then the scanf function will never read an entire string as a whitespace character occurs between the two names. The scanf function will only read GIST.

In order to read a string contains spaces, we use the gets() function. Gets ignores the whitespaces. It stops reading when a newline is reached (the Enter key is pressed).For example:

```
#include <stdio.h>
int main( )
 {
char full_name[25];
printf("Enter your full name: ");
gets(full_name);
printf("My full name is %s ",full_name);
return 0;
}
```

Input and Output:

```
Enter your full name: Dennis Ritchie
My full name is Dennis Ritchie
```

2d-array of charecters

The array of characters is called a string. “Hi”, “Hello”, and e.t.c are the examples of String. Similarly, the array of Strings is nothing but a two-dimensional (2D) array of characters. To declare an array of Strings in C, we must use the char data type
Declaration of the array of strings

Syntax:-

```
char string-array-name[row-size][column-size];
```

Here the first index (row-size) specifies the maximum number of strings in the array, and the second index (column-size) specifies the maximum length of every individual string.

Example of two dimensional characters or the array of Strings is,

```
char language[5][10] = {"Java", "Python", "C++", "HTML", "SQL"};
```

J	a	v	a	\0					
P	y	t	h	o	n	\0			
C	+	+	\0						
H	T	M	L	\0					
S	Q	L	\0						

TOPIC 12: STRING HANDLING FUNCTIONS

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

Function	Syntax (or) Example	Description
strcpy( )	strcpy(string1, string2)	Copies string2 value into string1
strlen( )	strlen(string1)	returns total number of characters in string1
strcat( )	strcat(string1,string2)	Appends string2 to string1
strcmp( )	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strlwr( )	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr( )	strupr(string1)	Converts all the characters of string1 to upper case.
strrev( )	strrev(string1)	It reverses the value of string1
strstr( )	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1

TOPIC 13:COMMAND LINE ARGUMENTS

The arguments passed from command line to main( ) function are called command line arguments. These arguments are handled by main( ) function.

To support command line argument, you need to change the structure of main( ) function as given below.

int main(int argc, char *argv[ ])

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[ ]** contains the total number of arguments. The first argument is the file name always.

Example Program 1:-

```
#include <stdio.h>
void main(int  argc, char *argv[ ] )
{
 int  i;
 printf("total no.of arguments=%d",argc);
 printf("\n List of arguments are:\n");
 for(i=1;i<argc;i++)
 {
        Printf("%s",argv[i]);
 }
}
```

Example program 2:- atoi( ) is a library function that converts string to integer

```
#include <stdio.h>
int main(int argc, char *argv[ ])
{
        int a,b,sum;
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        sum = a+b;
        printf("Sum of %d, %d is: %d\n",a,b,sum);
        return 0;
}
```

---*THE END***----**