Strings

String: String is defined as “group of characters, digits and special symbols enclosed between double quoted marks”.

1. Strings are nothing but array of characters ended with null character **‘\0’**. It indicates end of the string.

2. Strings are always enclosed by double quoted marks, whereas characters are enclosed by single quoted marks.

3. The control string “**%s”** is used to read and print string constants.

Declaration of Strings: String declaration can be by using arrays and pointers as follows.

Using arrays: **Syntax:** data_type array_name[index];

Example: char name[10];

Using pointers: **Syntax:** data_type *array_name;

Example: char *variable;

Initialization of strings: In C, string initialization can be done in following ways.

```
char c[]="abcd";

(or)

char c[5]="abcd";

(or)

char c[]={'a','b','c','d','\0'};

(or)

char c[5]={'a','b','c','d','\0'};
```

String can also be initialized using pointers as follows.

```
char *var=”abcd”;
```

NOTE: Difference between above initializations are, when we declare char as “**c[5]**“, 5 bytes of memory space is allocated for holding the string value, whereas when we declare char as “**c[]**”,memory space will be allocated as per the requirement during execution of the program.

Reading Strings from user

Reading word from user:

```
char name[20];
scanf("%s",&name);
```

String variable **name** can only take a word. It is because when white space is encountered, the scanf() function terminates.

C program to illustrate how to read string from terminal:

```
#include <stdio.h>
int main(){
        char name[20];
        printf("Enter name:");
        scanf("%s",&name);
        printf("Your name is %s.\n",name);
        return 0;
        }
```

Output:

```
Enter name: Narendra Modi
Your name is Narendra.
```

In above, program will ignore Modi because, scanf() function takes only string before the white space.

C program to illustrate how to read line of text without using string functions:

```
#include <stdio.h>
int main(){
        char text[30],ch;
        int i=0;
        printf("Enter line of text: ");
 while(ch!='\n') // terminates if user press Enter Key
        {
         ch=getchar();
        text[i]=ch;
        i++;
        }
name[i]='\0'; // inserting null character at end
printf("U have entered: %s",text);
return 0;
}
```

Output:

```
Enter line of text: Hi hello, very good morning to all.
U have entered: Hi hello, very good morning to all.
```

Reading or printing line of text:

There are predefined functions gets() and puts(), to read and print line of text as follows.

//C program to illustrate how to read and print, line of text using gets and puts functions:

```
#include<stdio.h>
int main(){
    char text[30];
    printf("Enter line of text:");
    gets(text);     //Function to read string from user.
    printf("U have entered:");
    puts(text);    //Function to display string.
    return 0;
}
```

Output:

```
Enter line of text: Hi hello, very good morning to all.
U have entered: Hi hello, very good morning to all.
```

String library functions

The string library **#include<string.h>** or **#include<strings.h>** provides some useful functions for working with strings, like for finding length of the string, joining of two strings, comparison of two strings, reversing of string etc. The following are some of these string operations.

1. strcpy(): This library function is used to copy a string of one variable to another.

Syntax: strcpy(string1, string2);

Example:
```
char string1[]=”abcd”;
char string2[]=”defghijk”;
strcpy(string1,string2); //string1 becomes defghijk
```

2. strcat(): This library function is used to join two strings.

Syntax: strcat(string1, string2);

Example:
```
char string1[]=”abcd”;
char string2[]=”defghijk”;
strcat(string1,string2); //string1 becomes abcddefghijk
```

3. strrev(): This library function is used to reverse the string.

Syntax: strrev(string1);

Example:
```
char string1[]=”abcd”;
strrev(string1); //string1 becomes dcba
```

4. strcmp(): This library function is used to compare two strings. It return 0 if two strings are equal, returns +ve value if ASCII value of first mismatching element of first string is greater than that of second string and -ve value if ASCII value of first mismatching element of first string is less than that of second string.

Syntax: strcmp(string1,string2);

Example:
```
char string1[]=”abcd”;
Char string2[]=”abcd”
strcmp(string1,string2); //it returns 0
```

5. strlen(): This library function is used to find length of the string.

Syntax: strlen(string1);

Example:
```
char string1[]=”abcd”;
strlen(string1); //it returns 4
```

6. strlwr(): This library function is used to convert uppercase alphabets to lowercase alphabets.

Syntax: strlwr(string1);

Example:
```
char string1[]=”ABCD”;
strlwr(string1); //string1 becomes abcd
```

7. strupr(): This library function is used to convert lowercase alphabets to uppercase alphabets.

Syntax: strupr(string1);

Example:
```
char string1[]=”abcd”;
strupr(string1); //string1 becomes ABCD
```

Functions

Definition: A function is a finite set of instructions that together perform a specific task. A large program is divided into basic building blocks called function. C function contains set of instructions enclosed between "**{ }**" which performs specific operation in a program.

A C program has at least one function called `main( )`. Without main() function, technically there is no possibility to write C program.

There are three aspects in each C function. They are:

- **Function declaration or prototype**: This statement informs to compiler about the function name, function parameters and return value's data type.
- **Function call:** This statement will make a call function definition.
- **Function definition**: This contains all the statements to be executed.

S. No	C function aspects	Syntax
1	function declaration	return_type function_name (argument list);
2	function call	function_name (arguments list);
3	function definition	return_type function_name (arguments list) { Body of function; }

Advantages of functions

1. It provides modularity to the program.
2. Code Reusability i.e. if we want to perform same task repetitively in another programs, then the function defined can be used for any number of times to perform the task.
3. Functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
4. It increases readability of program.

Function Types

Functions are classified into two types. They are,

1. Library Functions (or) built-in functions
2. User defined Functions

1. Library functions: Library functions are the in-built function in C programming system. The programmers can access or use these functions, but they cannot modify them.

For example: **printf():** prinf() is used for displaying output in C.

scanf(): scanf() is used for taking input in C.

2. User defined functions: C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions.

Suppose, a programmer wants to find factorial of a number and to check whether a number is prime or not in same program. We can create two separate user-defined functions in the program; one for finding factorial and other for checking whether a number is prime or not.

Example for user defined function:

```
#include<stdio.h>
int fact(int); //function prototyping or declaration
void main(){
int n, factorial;
printf("Enter a value to find its factorial:");
scanf("%d",&n);
factorial=fact(n);  // calling function
printf("factorial of %d=%d\n",n,factorial);
}
int fact(int n)   // function definition or called function
{
int i,x=1;
for(i=1;i<=n;i++){
x=x*i;
}
return x;   // retruns value to calling function
}
```

Types of User defined functions

Based on return type and arguments in functions, user-defined functions can be categorized as:

1. Function with no arguments and no return value.
2. Function with no arguments and return value.
3. Function with arguments but no return value.
4. Function with arguments and return value.

1. Function with no arguments and no return value: Here, keyword "**void**" means, it used to return a value to the calling function and there are no arguments contained in the pair of parenthesis of caller or called functions

Syntax:

```
void function_name(); // function declaration
function_name();          // function call
void function_name()      // function definition
{
    statements;
}
```

For example: To find “Area of Circle” using function with no arguments and no return value

```
#include<stdio.h>
void area();  // function prototyping
void main()
{
  area(); // function calling
}

void area() // function definition
{
   float area,radius;
   printf("Enter the radius : ");
   scanf("%f",&radius);
   area = 3.14 * radius * radius ;
   printf("Area of Circle = %f",area);
}

Output :

Enter the radius : 3
Area of Circle = 28.260000
```

2. Function with no arguments and return value: Here, keyword “**int**” means, it is used to return a value to the calling function and there are no arguments contained in the pair of parenthesis of caller or called functions.

Syntax:

```
int function_name();  // function declaration
function_name();      // function call
int function_name()   // function definition
{
    statements;
    return value;      // retruning value
}
```

For example: To find sum of two numbers using function with no arguments and return value

```
#include<stdio.h>
int sum(); // function prototyping
void main()
{
   int addition;
   addition = sum(); // function calling
   printf("Sum of two given values = %d", addition);
}

int sum()  // function definition
{
     int a = 50, b = 80, z;
     z = a + b;
     return z;
}

Output:
Sum of two given values = 130
```

3. Function with arguments but no return value: Here, keyword "**void**" means, it does not return anything to the calling function and there are arguments contained in the pair of parenthesis of caller or called functions.

Syntax:

```
void function_name( int);  // function declaration
function_name(variable);       // function call
void function_name(int variable)    // function definition
{
    statements;
}
```

For example: To find area of circle using function with arguments but no return value

```
#include<stdio.h>
void area(float rad);  // function prototyping
void main()
{
  float radius;
  printf("Enter the radius : ");
  scanf("%f",&radius);
  area(radius); // function calling
}

void area(float radius) // function definition
{
   float area;
   area = 3.14 * radius * radius ;
   printf("Area of Circle = %f",area);
}

Output :

Enter the radius : 3
Area of Circle = 28.260000
```

4. Function with arguments and return value: Here, keyword "**int**" means, it used to return a value to the calling function and there are arguments contained in the pair of parenthesis of caller or called functions.

Syntax:

```
int function_name( int);  // function declaration
function_name(variable);       // function call
int function_name(int variable)    // function definition
{
    statements;
    return value;       // retruning value
}
```

For example: To find square of a number using function with arguments and return value

```
#include<stdio.h>
float square ( float x ); //function prototyping
void main( )
{

    float m, n ;
    printf ( "Enter number to find square:");
    scanf ( "%f", &m ) ;
    // function call
    n = square ( m ) ; //function calling
    printf ( "\nSquare of the given number %f is %f\n",m,n );
}

float square ( float x )  // function definition
{
    float p ;
    p = x * x ;
    return ( p ) ;
}

Output:
Enter some number for finding square:2
Square of the given number 2.000000 is 4.000000
```

NOTE

- The return and arguments type can be of any data type, available in C programming language.
- The keyword "**void**" is used to return nothing (empty), and we can also use void for empty arguments (optional).

exit() Statement

- **exit()** is used to exit the program as a whole. In other words it returns control to the operating system.
- After **exit()** all memory and temporary storage areas are all flushed out and control goes out of program.
- **exit()** statement is placed as the last statement in a program since after this, program is totally exited.

  Syntax: void exit(int status);

return Statement

- The **return()** statement is used return a value to the calling function and assigns to the variable in the left side of the calling function.
- If a function does not return a value, the return type in the function declaration and definition is specified as void.
- In contrast **return()** statement can take its presence anywhere in the function. It need not be presented as the last statement of the function.

Syntax: return (expression); (or) **return** value;

Arguments and Parameters

Arguments and parameters refer to the variables or expressions passed from the caller of a function to the function definition or a variable or expressions returned from function definition to called function.

- **Argument:** that which is passed into a function by its caller i.e. variables held by calling function.
- **Parameter:** that which is received by the function i.e. variable held or received by function definition.

NOTE:

1. Parameters are simply variables.
2. Syntactically we can pass any number of parameters to functions.
3. Parameters are specified within pair of parenthesis.

Types of Parameters

1. **Actual Parameters:** Parameters written in calling function are called “Actual Parameters”.

```
#include<stdio.h>
void display(int); //function prototyping
void main()
{
int a=10;
display(a); // calling function holding actual parameter ‘a’
}

void display(int a) // function definition or called function
{
```

```
printf(“value received from calling function=%d”, a);
}
```

2. **Formal Parameters:** Parameters written in function definition are called **“Formal Parameters”.**

```
#include<stdio.h>
void display(int); //function prototyping
void main()
{
 int a=10;
 display(a); // calling function holding actual parameter ‘a’
}

void display(int a) // function definition holding formal parameter ‘a’
{
 printf(“value received from calling function=%d”, a);
}
```

Types of Function Call’s: Functions are called by their names. If the function is without argument, it can be called directly by using its name. But for functions with arguments, we have two ways to call them. They are:

1. Call by Value
2. Call by Reference

1. Call by Value: In this calling technique, we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged, only the parameters inside function can be modified.

```
#include<stdio.h>
void calculate(int);
void main()
{
 int a=10;
 display(a); // calling function holding actual parameter ‘a’
 printf(the value after function execution=%d”,a); //a=10
}

void display(int a) // function definition holding formal parameter ‘a’
{
 a=a+10;
}
```

Output: the value after function execution=10

In this case the actual variable ‘**a**’ is not changed, because we pass argument by value, hence a copy of ‘**a**’ is passed, which is changed, and that

copied value is destroyed as the function ends (goes out of scope). So the variable '**a**' inside main() still has a value 10.

2. Call by Reference: In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer; in both the case they will change the values of the original variable i.e. the changes made to the formal parameter s affect the actual parameters.

```
#include<stdio.h>
void swap(int *a, int *b);
void main()
{
 int a=10,b=20;
 swap(&a,&b);
 printf("values after function execution a=%d b=%d",a,b); // a=20 b=10
}

void swap(int *a, int *b)
{
 int temp;
 temp=*a;
 *a=*b;
 *b=temp;
}
```

Differences between call by value and call by reference:

S. No	Call by Value	Call by Reference
1	Duplicate copy of original Parameter is passed to function definition	Actual copy of original parameter is passed to function definition.
2	No effect on original parameter after modifying parameter in function	Original parameter gets affected if value of parameter changed inside function definition

Recursion

A function that calls itself is known as recursive function and this technique is known as recursion in C programming language.

Merits:

- Avoidance of unnecessary calling of functions.
- Extremely useful when applying the same solution.
- Using recursive function can reduces the size of the code.

Demerits:

- A recursive function is often confusing.
- The exit point must be explicitly coded.
- It is difficult to trace the logic of the function.

Examples of C programs using recursive functions

1. Write a C program to find sum of first n natural numbers using recursion.

```
#include<stdio.h>
int sum(int);
void main()
{
 int add,n;
 printf(Enter positive integer:”);
 scanf(“%d”,&n);
 add=sum(n);
 printf(Total sum of First %d natural numbers=%d”,n,add);
}

int sum(int n)
{
 if(n==0)
  return n;
 else
  return n+sum(n-1);
}
```

Output

```
Enter a positive integer: 5
Total sum of First 5 natural numbers=15
```

2. Write a C program to print N Fibonacci numbers using recursion.

```
#include<stdio.h>
int fibonacci(int);
void main()
{
 int n,c=0,i,fib;
 printf(“Enter value of n:”);
 scanf(“%d”,&n);
 printf(“Fibonacci Series\n”);
 for(i=1;i<=n;i++)
 {
  fib=fibonacci(c);
  printf(“%d\t”,fib);
  c++;
 }
}

int fibonacci(int n)
{
 if(n==0)
  return ;
 else if(n==1)
  return 1;
 else
  return (fibonacci(n-1)+fibonacci(n-2));
}
```

Output

```
Enter value of n:9
Fibonacci Series
0 1 1 2 3 5 8 13 21
```

3. Write a C program to find factorial of a number using recursion.

```
#include<stdio.h>
int fact(int);
void main()
{
 int n,fact;
 printf(“Enter a positive integer:”);
 scanf(“%d”,&n);
 fact=factorial(n);
 printf(“Factorial of %d is %d\n”, n,fact);
}

int factorial(int n);
{
 if(n==0)
  return 1;
 else
  return n*factorial(n-1);
}
```

Output

```
Enter a positive integer:6
Factorial of 6 is 720
```

Command Line arguments

Passing arguments from the command line to your C programs are called **command line arguments.**

- These command line arguments are supplied to a program when the program is invoked.
- Many times they are important for the program, especially when we want to control the program from outside instead of hard coding those values inside the code.

The command line arguments are handled using **main()** function arguments and actual syntax of main() is as follows.

Syntax: `int main(int argc, char *argv[])`

- where **argc** is an argument counter, refers to the number of arguments passed and
- **argv[]** is a argument vector, it is a pointer array which points to each argument passed to the program.
- **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument.
- If no arguments are supplied, **argc** will be **1**, and if you pass one argument then **argc** is set at 2.

Example:

```
#include<stdio.h>
int main(int argc, char *argv[])
{
   if(arc==2){
         printf(“The argument supplied is %s\n”,arg[1]);
     }
      else if(argc>2){
         printf(“Too many arguments supplied.\n”);
       }
   else{
      printf(“One argument expected.\n”);
   }

}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out testing1 testing2
One argument expected.
```

Arrays

Definition: An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

Here the words,

- **Finite** means data range must be defined.
- **Ordered** means data must be stored in continuous memory addresses.
- **Homogenous** means data must be of similar data type.

Characteristics:

- All the elements of an array share the same name.
- Array elements are shared using index. And index value starts from **zero**.
- The memory allocated for an array is continuous.
- The amount memory required for holding elements in array depends on its type and size i.e. **total bytes = size of data type * size of array**.

Advantages:

- It is used to represent multiple data items of same type by using only single name.
- It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.
- 2D arrays are used to represent matrices.

Disadvantages:

- We must know in advance that how many elements are to be stored in array.
- Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.

- Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted.

- The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

Index or Subscript variable

1. Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript variable / index i.e. tells the range of an array.

2. Subscript Variables helps us to identify the item number to be accessed in the contiguous memory.

Types of Arrays

1. 1D or single dimensional arrays

2. 2D or two dimensional arrays

Single Dimensional Arrays

1. Single or One Dimensional array is used to represent and store data in a linear form.

2. Array having only one subscript variable is called **One-Dimensional array**

3. It is also called as **Single Dimensional Array** or **Linear Array**

Array Declaration: To declare an array in C, a programmer has to specify the type of the elements and the number of elements required by an array.

Syntax: data_type array_name[size]; **Example:** int arr[10];

arr **[0]** **[1]** **[2]** **[3]** **[4]** **[5]** **[6]** **[7]** **[8]** **[9]**

Here **int** is the data type, **arr** is the name of the array and 10 is the size of array. It means array **arr** can only contain 10 elements of **int** type. **Index** of an array starts from **0** to **size-1** i.e. first element of **arr** array will be stored at arr[0] address and last element will occupy arr[9].

Array Initialization: After an array is declared it must be initialized. Otherwise, it will contain **garbage** value (any random value). An array can be initialized at either **compile time** or at **runtime**.

Compile time Array initialization: Compile time initialization of array elements means giving values at the time array declaration. The general form of initialization of an array is,

Syntax: data_*type* **array_name[size] = { list of values };**

Example: int arr[10]= { 5, 7, 8, 9, 2, 1, 3, 4, 6, 10 };

	5	**7**	**8**	**9**	**2**	**1**	**3**	**4**	**6**	**10**
arr	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**

Note: Giving more values than array size leads to compile time error

Runtime Array initialization: An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values. Example,

```
#include<stdio.h>
void main()
{
        int arr[4],i;
        printf("Enter array element");
        for(i=0;i<4;i++)
        {
                scanf("%d",&arr[i]);   //Run time array initialization
         }

        for(i=0;i<4;i++)
        {
                printf("%d\n",arr[i]);
         }
}
```

One dimensional array and functions

Array elements can be passed to functions in two ways. They are:

1. Passing single element of an array to function.
2. Passing total array to function.

1. Passing single element of an array to function:

```
#include<stdio.h>
void display(int);
void main()
{
        int arr[]={10,20,30,40,50},i;
        for(i=0;i<5;i++)
        {
                diplay(arr[i]);
         }
}

void display(int x){
                printf(“%d\t”,x);
}
```

Output: 10 20 30 40 50

2. Passing total array to function:

```
#include<stdio.h>
void display(int x[],int m);
void main()
{
        int arr[]={10,20,30,40,50},i;
        diplay(arr,5);
}

void display(int x[], int m){
        int j;
        for(j=0;j<m;j++){
           printf("%d\t",x[j]);
        }
}
```

Output: 10 20 30 40 50

Two Dimensional Arrays

1. Array having two subscript variables is called Two-Dimensional array.

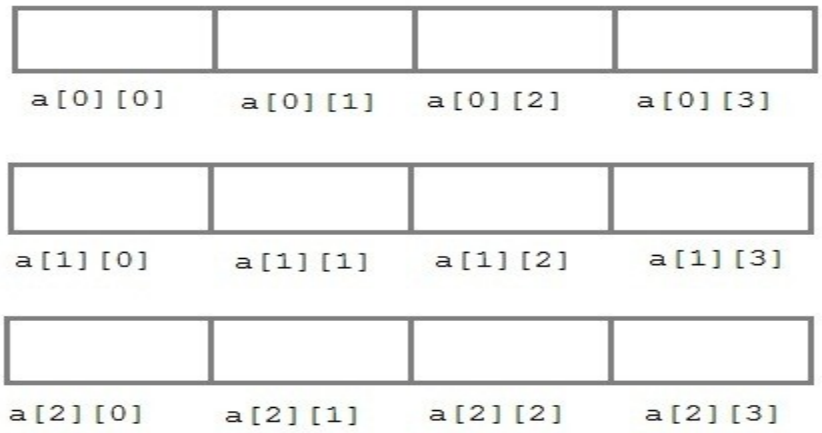
2. Two dimensional Arrays are looks like **Matrix** form.

3. C language supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

4. Memory is allocated contiguously, and number of bytes required for two dimensional arrays= no. of rows * no. of columns * size of data type.

   Syntax: data_type array_name[row_size][column_size];

   Example: int a[3][4];



The above array can also be declared and initialized together. Such as,

```
int arr[][4] = {  {1,2,3,4},{2,3,4,5},{3,4,5,6} };
```

Accessing Array Elements

We access the array elements directly using their **indices**. Each element can be accessed through the name of the array and the element's **index** placed in the brackets. For example,

```
double salary = balance[9]; //single dimensional array
```

The above statement will take the **10th** element from the array and assign the value to salary variable.

```
double salary [0][1]=balance[4][2]; // two dimensional array
```

The above statement will take the **4th** row **2nd** column positioned element array balance and assigns to **0th** row **1st** column position of array salary.

Two dimensional arrays and functions

Two dimensional arrays can be passed to function in two ways:

1. Passing single element of an array to function
2. Passing total array to the function

1. Passing single element of an array to function:

In order to access single element of two dimensional arrays, both row index and column index should be specified.

```
#include<stdio.h>
void display(int);
void main()
{
        int arr[2][2]={10,20,30,40},i,j;
        for(i=0;i<2;i++){
           for(j=0;j<2;j++){
                diplay(arr[i][j]);
           }
         }
}

void display(int x){
               printf("%d\t",x);
}
```

Output: 10 20 30 40

2. Passing total array to the function:

```
#include<stdio.h>
void display(int x[][], int m, int n);
void main()
{
        int arr[2][2]={10,20,30,40};
        display(arr,2,2);

}

void display(int x[][], int m, int n){
        int i,j;
        for(i=0;i<m;i++){
           for(j=0;j<n;j++){
                printf(“%d\t”,x[i][j]);
           }
         }
}
```

Output: 10 20 30 40

Structures and Unions

We know that arrays can be used to represent a group of data items that belong to the same type, such as int or float. However we cannot use an array if we want to represent a collection of data items of different types using a single name. A structure is a convenient tool for handling a group of logically related data items.

Structure: Structure is composition of the different variables of different data types, grouped under same name.

```
Syntax: struct struct_name{
                datatype1 var1;
                datatype2 var2;
                datatype3 var3;
          };
```

```
Example: struct student{
              char name[64];
               char course[12];
               int age;
              int year;
             };
```

Some Important Definitions of Structures:

1. Each variables declared in Structure is called **member.**

```
char name[64];
char course[128];
int age;
int year;
```

   are some examples of members.

2. Name given to structure is called as **tag**

```
student
```

3. Structure **member** may be of **different data type** including **user defined data-type** also

```
typedef struct {
          char name[64];
          char course[128];
          book b1;
          int year;
  } student;
```

Here, **book** is user defined data type.

NOTE

1. **Struct** keyword is used to declare structure.
2. **Members of structure** are enclosed within opening and closing braces.
3. **Declaration** of Structure reserves **no space** in memory.
4. Structure declaration is nothing but the "**Template / Map / Shape"** of the structure.
5. Memory is created, very first time when the **variable is created** / **Instance** is created.

Different Ways of Declaring Structure Variable:

Way 1: Immediately after Structure Template

```
struct date
{
     int date;
     char month[20];
     int year;
}today;

// "today" is name of Structure variable
```

Way 2: Declare Variables using struct Keyword

```
struct date
{
    int date;
    char month[20];
    int year;
};

struct date today;
```

Where "**date**" is name of structure and "**today**" is name of variable.

Way 3: Declaring Multiple Structure Variables

```
struct Book
{
    int pages;
    char name[20];
    int year;

} book1, book2, book3;
```

Initializing structure elements

```
#include<stdio.h>
struct emp
{
char name[20];
int empno;
float sal;
};

void main()
{
struct emp e1={“Rahul”,10,20000};
struct emp e2={“Rani”};
struct emp e3,e4;
e3=e1;
e4.sal=e1.sal;
printf(“Employee 1 details\n”);
printf(“%s%d%f\n”,e1.name,e1.empno,e1.sal);
printf(“Employee 2 details\n”);
printf(“%s%d%f\n”,e2.name,e2.empno,e2.sal);
printf(“Employee 3 details\n”);
printf(“%s%d%f\n”,e3.name,e3.empno,e3.sal);
printf(“Employee 4 details\n”);
printf(“%s%d%f\n”,e4.name,e4.empno,e4.sal);
}
```

Output:

```
Employee 1 details
Rahul 10 20000
Employee 2 details
Rani 0 0.0
Employee 3 details
Rahul 10 20000
Employee 4 details
NULL 0 20000
```

- In the above program, for **e1** values are directly given at the time of declaration.
- For **e2**, only value is given. When one or more initializes are missing the structure elements will be assigned to default values.

For string - default value is NULL
For int - default value is zero
For float - default value is 0.0

- The values of **e3** is copied from **e1** using assignment statement **e3=e1**; here, all three values of **e1** copied to **e3**.
- It is also possible to copy the single element of one structure variable to another variable of same structure, **e4.sal=e1.sal**; here, only **sal** value is copied.

Reading structure members from the user

The structure member values can be read from the user using scanf() function as follows.

```
#include<stdio.h>
struct emp
{
  char name[20];
  int empno;
  float sal;
}e1;

void main()
{
  printf("Enter employee 1 details\n");
  scanf("%s%d%f",&e1.name, &e1.empno, &e1.sal");
  printf("Employee 1 details are\n");
  printf("%s%d%f\n",e1.name,e1.empno,e1.sal);
}
```

Output:

```
Enter employee 1 details
 Rahul 10 20000
 Employee 1 details are
 Rahul 10 20000
```

Array of structures

- Array is a collection of similar data types. In the same way we can also define array of structures.
- In array of structures, every element of array is structure type.

```
#include<stdio.h>
struct emp
{
  char name[20];
  int empno;
  float sal;
};
```

```
void main()
{
struct emp e[10];
int i;
for(i=0;i<10;i++){
  printf(\n enter %d employee details”,i+1);
  scanf(“%s%d%f”,&e[i].empname, &e[i].empno, &e[i].sal);
  }
for(i=0;i<10;i++){
  printf(“%d employee details are\n”);
  printf(“%s%d%f\n”,e[i].name,e[i].empno,e[i].sal);
 }
}
```

In the above program array of 10 **emp** variables are created that means 10*16 = 160 bytes of memory allocated.

Pointer to structure

- Pointer is a variable, which is used to store the address of another variable of same type.
- Pointer is a variable, which is used to store the address of another variable of same type.
  Syntax: structure_name *variable_name;
  Example: Struct **emp** *p;
- In order to access the members of structure ‘**->**’ operator is used.

```
struct emp
{
   char name[20];
   int empno;
   float sal;
};

void main()
{
   struct emp *p;
   struct emp e1={“Rahul”,10,20000};
   p=&e1;
   printf(“Employee details are\n”);
   printf(“%s%d%f\n”,p->name,p->empno,p->sal);
}
```

Output:

```
Employee details are
Rahul 10 20000
```

Structures and functions

Structures can be passed to functions in two ways as follows.

1. Passing single element of structure
    a) Passing single element of structure using call by value
    b) Passing single element of structure using call by reference
2. Passing total structure
    a) Passing total structure to function using call by value
    b) Passing total structure to function using call by reference

1. Passing single element of structure

a) Passing single element of structure using call by value

```
struct emp
{
  int hours;
  int minutes;
  int seconds;
};

void display1(int x);
void display2(int x);
void display3(int x);

void main()
{
  struct time t1={10,20,31};
printf(“Time at outside main\n”);
  display1(t1.hours);
  display2(t1.minutes);
  display3(t1.seconds);
printf(“Time in main function\n”);
printf(“%d:%d:%d\n”, t1.hours,t1.minutes,t1.seconds);
}

void display1(int x)
{
  x=x+2;
  printf(“%d:”,x);
}

void display2(int x)
{
  x=x+20;
  printf(“%d:”,x);
}

void display3(int x)
{
  x=x+10;
  printf(“%d\n”,x);
}
```

Output:

```
Time at outside main
  12:40:41
Time in main function
  10:20:31
```

b) Passing single element of structure using call by reference

```
struct emp
{
    int hours;
    int minutes;
    int seconds;
};

void display1(int *x);
void display2(int *x);
void display3(int *x);

void main()
{
    struct time t1={10,20,31};
printf("Time at outside main\n");
    display1(t1.hours);
    display2(t1.minutes);
    display3(t1.seconds);
printf("Time in main function\n");
printf("%d:%d:%d\n", t1.hours,t1.minutes,t1.seconds);
}

void display1(int *x)
{
    *x=*x+2;
    printf("%d:", *x);
}

void display2(int *x)
{
    *x=*x+20;
    printf("%d:",*x);
}

void display3(int *x)
{
    *x=*x+10;
    printf("%d\n", *x);
}
```

Output:

```
Time at outside main
    12:40:41
Time in main function
    12:40:41
```

2. Passing total structure

a) Passing total structure to function using call by value

```
struct emp
{
      int hours;
      int minutes;
      int seconds;
};

void display(struct time t);

void main()
{
struct time t1={10,20,31};
printf(“Time at outside main\n”);
display1(t1);
printf(“Time in main function\n”);
printf(“%d:%d:%d\n”, t1.hours,t1.minutes,t1.seconds);
}

void display1(struct time t)
{
      t.hours=t.hours+10;
      t.minutes=t. minutes+10;
      t.seconds=t.seconds+10;
      printf(“%d:%d:%d\n”,t.hours,t.minutes,t.seconds);
}
```

Output:

```
Time at outside main
      20:30:41
Time in main function
      10:20:31
```

b) Passing total structure to function using call by reference

```
struct emp
{
      int hours;
      int minutes;
      int seconds;
};

void display(struct time *t);

void main()
{
struct time t1={10,20,31};
printf(“Time at outside main\n”);
display1(&t1);
printf(“Time in main function\n”);
printf(“%d:%d:%d\n”, t1.hours,t1.minutes,t1.seconds);
}

void display1(struct time t)
{
      t->hours=t->hours+10;
      t->minutes=t-> minutes+10;
      t->seconds=t->seconds+10;
      printf(“%d:%d:%d\n”,t->hours,t->minutes,t->seconds);
}
```

Output:

```
Time at outside main
      20:30:41
Time in main function
      20:30:41
```

Structure within structure (or) Self referential structures

Nested structure is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure. The structure variables can be a normal structure variable or a pointer variable to access the data.

Structure within structure using normal structure variable:

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

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

void main()
{
  struct student_detail stu_data = {1, "Raju", 90.5, 71145,"Anna
University"};
 printf(" Id is: %d \n", stu_data.id);
printf(" Name is: %s \n", stu_data.name);
 printf(" Percentage is: %f \n\n", stu_data.percentage);
 printf(" College Id is: %d \n",  stu_data.clg_data.college_id);
printf(" College Name is: %s \n", stu_data.clg_data.college_name);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 90.500000

College Id is: 71145
College Name is: Anna University
```

Structure within structure using pointer variable:

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

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

void main()
{
 struct student_detail stu_data = {1, "Raju", 90.5, 71145,"Anna
University"};
 stu_data_ptr = &stu_data;
printf(" Id is: %d \n", stu_data_ptr->id);
printf(" Name is: %s \n", stu_data_ptr->name);
printf(" Percentage is: %f \n\n",  stu_data_ptr->percentage);
 printf(" College Id is: %d \n",  stu_data_ptr->clg_data.college_id);
printf(" College Name is: %s \n", stu_data_ptr-
>clg_data.college_name);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 90.500000

College Id is: 71145
College Name is: Anna University
```

Union

Union is a collection of different data types, but it holds one variable at a time.

- In case of structure each member has its own memory i.e. for all data type variables, but in unions it allocates memory for only one variable i.e. number of bytes required for the largest member.
- In unions one memory location is shared by all the members of union.

NOTE

✓ To access union members, it follows the same syntax that we use for structure members.

Comparisons between Structure and Union

Structure	Union
It allocates memory, each individual member in the structure.	It allocates piece of memory, for number of bytes required for the largest member.
Each member has their own memory space.	One block is used by all the members of union.
Structure cannot be implemented in shred memory.	Union is the Best environment where memory is shared.
It has less Ambiguity.	As memory is shared, Ambiguity is more in union.
Self referential structure can be implemented in data structure.	Self referential union cannot be implemented.
All members of the structure can be accessed at a time.	Only one member is accessed at a time.

typedef keyword

It allows us to specify a new name for a data type which already provided by C compiler.

Syntax: typedef data_type new_name;

```
void main(){
      typedef int num;
      num i=10,j=20;
      printf(“%d != %d”i,j);    // 10 != 20
}
```

Pointers

Definition: A **pointer** is a variable whose value is the address of another variable of same data type i.e. direct address of the memory location. Like any variable or constant, you must declare a **pointer** before using it to store any variable address.

The general form of a pointer variable declaration is:

Syntax: data_type *variable_name;

Here, **data_type** is the pointer's basic data type; it must be a valid C data type and **variable-name** is the name of the pointer variable. The asterisk ***** used to declare a pointer variables and the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer variable.

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to pointer variable. Pointer variable contains address of variable of same data type. In C language address operator **&** is used to determine the address of a variable. The **&** (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ;        //pointer declaration
ptr = &a ;        //pointer initialization
or,
int *ptr = &a ;      //initialization and declaration together
```

Pointer variable always points to same type of data.

```
float a;
int *ptr;
ptr = &a;    //ERROR, type mismatch
```

Dereferencing of Pointer (*)

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using the **indirection operator** *.

```
int a,*p;
a = 10;
p = &a;
printf("%d",*p);    //this will print the value of a.
printf("%d",*&a);  //this will also print the value of a.
printf("%u",&a);  //this will print the address of a.
printf("%u",p);  //this will also print the address of a.
printf("%u",&p);  //this will also print the address of p.
```

Advantages

- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers allow us to perform dynamic memory allocation and de-allocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- Pointers allow us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers.

Disadvantages

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Basically, pointer bugs are difficult to debug. Its programmer's responsibility to use pointers effectively.

NULL Pointers

It is always a good practice to assign a **NULL** value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned **NULL** is called a **null** pointer.

The **NULL** pointer is a constant with a value of zero defined in several standard libraries. Consider the following program

```
#include <stdio.h>
void main(){
```

```
        int  *ptr = NULL;
        printf("The value of ptr is : %x\n", ptr  );
}
```

When the above code is compiled and executed, it produces the following result.

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

/* Source code to demonstrate, handling of pointers in C program */

```
#include <stdio.h>
int main(){
   int* pc;
   int c;
   c=22;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   pc=&c;
   printf("Address of pointer pc:%d\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   c=11;
   printf("Address of pointer pc:%d\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   *pc=2;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

Arithmetic Operations on Pointers

The operations that can be performed on pointers are:

1. Addition of a number to a pointer

Example:

```
int i=4, *j, *k;
j=&i;
j=j+1;// addition (2000+(1*2))=2002
k=j; // here k holds 2002
```

2. Subtraction of a number from a pointer

Example:

```
int i=4, *j, *k;
j=&i;
j=j-1;// addition (2000-(1*2))=1998
k=j; // here k holds 1998
```

3. Subtraction of one pointer from another pointer

Example:

```
int a[]={10,20,30,40,50,50};
int *i,*j,k;
i=&a[1];
j=&a[3];
k=j-i; // 2006-2004 = 2, held by k
```

4. Comparisons of two pointers

Example:

```
void main()
{
int a[]={10,20,30,40,50,50},*j,*k;
j=&a[4];
k=(a+4); // ‘a’ give starting address of array, 2000+(4*2)=2008
if(j==k)
   printf(“Equal”);
else
   printf(“Not equal”);
}
```

The operations that are not performed on pointers are:

1. Addition of two pointers
2. Multiplication of a pointer with constant
3. Division of a pointer with constant

Relation between Arrays and Pointers

For example,

int arr[10];

A value inside the address **&arr[0]** and address **arr** are equal. Value in address **&arr[0]** is **arr[0]** and value in address **arr** is ***arr**. Hence, **arr[0]** is equivalent to ***arr**.

Similarly, we can use any of the following based on our requirement in the program.

&a[1] is equivalent to **(a+1)** and, **a[1]** is equivalent to ***(a+1)**.
&a[2] is equivalent to **(a+2)** and, **a[2]** is equivalent to ***(a+2)**.
&a[3] is equivalent to **(a+1)** and, **a[3]** is equivalent to ***(a+3)**.
.
.
&a[i] is equivalent to **(a+i)** and, **a[i]** is equivalent to ***(a+i)**.

//Program to find the sum of six numbers with arrays and pointers.

```
#include <stdio.h>
int main(){
  int i,class[6],sum=0;
  printf("Enter 6 numbers:\n");
  for(i=0;i<6;++i){
      scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]
      sum += *(class+i); // *(class+i) is equivalent to class[i]
  }
  printf("Sum=%d",sum);
  return 0;
}
```

Output:

```
Enter 6 numbers:

2
3
4
5
3
4
Sum=21
```

Pointers and Arrays

Array is a group of similar data type elements. So the compiler allocates memory for array continuously. Fox example, **int** a[5]; compiler allocates **10** bytes ofmemory continuously for array '**a**'

```
#include<stdio.h>
void main()
{
int a[5]={10,20,30,40,50};
int i,j;
for(i=0;i<5;i++){
  printf("\n address=%u",&a[i]);
  printf("\t value=%d",a[i]);
 }
}
```

The above program can be written as follows.

```
#include<stdio.h>
void main()
{
int a[5]={10,20,30,40,50};
int i;
for(i=0;i<5;i++){
  printf("\n address=%u",(a+i));
  printf("\t value=%d",*(a+i));
 }
}
```

Output: above two programs produces same results as

```
  address=2000  value=10
  address=2002  value=20
  address=2004  value=30
  address=2006  value=40
  address=2008  value=50
```

The above program using pointer variable as follows.

```
#include<stdio.h>
void main()
{
int a[5]={10,20,30,40,50};
int *p,i;
p=&a[0];  // or p=a
for(i=0;i<5;i++){
  printf("\n address=%u",p);
  printf("\t value=%d",*p);
  p++;
 }
```

```
}
```

Passing single element of an array to function using pointers

```
#include<stdio.h>
void display(int *n);
void main()
{
int a[]={10,20,30,40,50};
int i;
for(i=0;i<5;i++){
  display(&a[i]);
  }
for(i=0;i<5;i++){
  printf("%d",a[i]);
  }
}

void display(int *n){
        *n=*n+10;
  }
```

Output:

```
  20 30 40 50 60
```

Passing an entire array to function using pointers

```
#include<stdio.h>
void display(int *j,int n);
void main()
{
int a[]={10,20,30,40,50};
display(a,5);
}

void display(int *j,int n){
        int i=1;
        while(i<=n){
               printf("element=%d\n",*j);
               i++;
               j++;
         }
  }
```

Output:

```
  10 20 30 40 50
```

Pointers and two dimensional arrays

```
#include<stdio.h>
void main(){
int i,j,*p;
int a[3][3]={ {1,2,3},{4,5,6}.{7,8,9} };
printf("Elements of an array with their addresses are as
follows\");
p=&a[0][0];
for(i=0,j=0;i<9;i++,j++){
  printf("%d[%u]\t",*p,p);
  p++;
  if(j==3){
         printf("\n");
         j=0;
   }
   }
}
```

Output:

```
Elements of an array with their addresses are as follows
1[2000] 2[2002] 3[2004]
4[2006] 5[2008] 6[2010]
7[2012] 8[2014] 9[2016]
```

Pointers to pointers

Pointer to pointer is a variable which stores the address of pointer variable of same data type.

```
#include<stdio.h>
void main()
{
int *p,**p1;
int i=10;
p=&i;
p1=&p;
printf("I address=%u\n",p);
printf("I address=%u\n",*p1);
printf("I value=%d\n",*p);
printf("I value=%d\n",**p1);
printf("P address=%u\n",&p);
printf("P address=%u\n",p1);
}
```

Output:

```
I address=2000
```

```
I address=2000
I value=10
I value=10
P address=3000
P address=3000
```

Array of Pointers (or) Pointers Array

In C it is possible to create array of pointers i.e. each index in array points to another variable of same data type.

```
#include<stdio.h>
void main()
{
int *a[4];
int i=31,j=5,k=19,l=71,m;
a[0]=&i;
a[1]=&j;
a[2]=&k;
a[3]=&l;
for(m=0;m<4;m++){
  printf("%d\t",*(a[m]));
  }
}
```

Output:

```
  31 5 19 71
```

Dynamic memory allocation

The process of allocating memory during program execution is called **dynamic memory allocation.**

C language offers four dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

S.No	Function	Syntax
1	malloc()	malloc (number *sizeof(int));
2	calloc()	calloc (number, sizeof(int));
3	realloc()	realloc (pointer_name, number * sizeof(int));
4	free()	free (pointer_name);

1. malloc()

- This function is used to allocate space in memory during the execution of the program.
- It does not initialize the memory allocated during execution. It carries garbage value.
- It returns NULL pointer if it could not able to allocate requested amount of memory.

2. calloc()

- This function is also like malloc() function. But calloc() initializes the allocated memory to zero.

3. realloc()

- This function modifies the allocated memory size by malloc() and calloc () functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. free()

- This function frees the allocated memory by malloc(), calloc(), realloc() functions and returns the memory to the system.

Files

Definition

File is a collection of records which will be stored on the disk. These records can be accessed through the set of library functions. Files are of two types:

1. Text files
2. Binary files

Text files

- It is a file which stores the data in the form of characters.
- Each line in text file ends with '**\n'** (new line).
- Each file is ends with a special character called **EOF** (End of File).

Binary files

- Data are stored in the same format as they are stored in memory.
- Each line in binary files does not end with **'\n**'.
- There is an end of file marker **EOF** for each file.

Files operations

The various operations that are performed on files are:

1. Creating new file
2. Opening an existing file
3. Reading from file
4. Writing to file
5. Moving to specific location in a file
6. Closing a file

Fopen()

Fopen() is used to open the file in specific mode.

Syntax: FILE *fp;
fp=fopen(“filename”,”mode”);

File is a structure that is defined in standard IO library(**stdio.h**). Fopen() opens the file in given mode and returns address to ‘**fp**’.

Fopen() performs three important tasks (in read mode):

- Firstly it searches on the disk the file to be opened.
- Then it loads the file from the disk into a place in memory called buffer.
- It sets up a character pointer that point to the first character of the buffer.

Different modes of files

Mode(binary file)	Mode(text file)	Meaning
rb	r	This mode allows opening file for reading data from it.
wb	w	This mode allows opening file for writing data to it.
ab	a	This mode allows opening file for appending data at the end of the file.
r+b	r+	This mode allows opening file for reading, writing and modifying existing data to file.
w+b	w+	This mode allows opening file for reading, writing and modifying existing data to file.
a+b	a+	This mode allows opening file for reading, writing and modifying existing data to file.

Fclose()

This function is used to close the opened file.

Syntax: fclose(FILE *);

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
fp=fopen(“abc.txt”,”r”);// returns address of first character
if(fp==NULL)
{
 printf(“\n File is not there”);
 exit(1);
}
printf(“\n file is successfully opened”);
fclose(fp);
getch();
}
```

Output:

```
Case 1: if file is there then output is “file is successfully opened”.
Case 2: if file is not there output is “file is not there”.
```

Read mode

In order to read the data from the file, following functions are used.

1. **fgetc():** Used to read single character from file.
2. **fgets():** Used to read group of characters from file.
3. **fscanf():** Used to read one record information from file.

In read mode;

Case 1: If file exist then it opens file and returns address of file to file pointer.
Case 2: If file not exist then fopen() returns NULL value.

1. fgetc(): This function is used to read single character from file. If fgetc() is used in program, it executed means, it performs two operations.

1. It reads character pointed by **fp**(file pointer).

2. And moves **fp** to next character.

Syntax: **char** fgetc(FILE *fp);

Example:

```
#include<stdio.h>
void main()
{
FILE  *fp;
char ch;
fp=fopen(“abc.txt”,”r”);
if(fp==NULL)
{
  printf(“file is not there”);
  exit(1);
}
ch=fgetc(fp);
printf(“%c”,ch);
ch=fgetc(fp);
printf(“%c”,ch);
fclose(fp);
getch();
}
```

Example: Program to read all the information present in file.

```
#include<stdio.h>
void main()
{
char ch;
FILE *fp;
fp=fopen(“abc.txt”,”r”);
if(fp==NULL)
{
  printf(“\n file is not there:”);
  exit(1);
}
printf(“\n file contents are:”);
while(1)
{
  ch=fgetc(fp);
  if(ch==EOF)
  break;
else
  printf(“%c”,ch);
}
fclose(fp);
getch();
}
```

2. fgets(): This function is used to read the group of characters from file.

Syntax: void fgets(char *strptr, int size, FILE *fp);

The function fgets() takes three arguments:

- The first is the address where the string is stored that is destination string.
- Second is the maximum length of the string to be readed from file.
- The third argument is the pointer to the structure FILE.

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
char s[80];
fp=fopen(“abc.txt”,”r”);
if(fp==NULL)
{
  puts(“cannot open file”);
  exit(1);
}
fgets(s,10,fp);
printf(“%s”,s);
fclose(fp);
getch();
}
```

3. fscanf(): fscanf() is used to read different data type values from the file.

Syntax: void fscanf(FILE *,”formatstring”,addresslist);

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
int a;
float b;
char x;
fp=fopen(“abc.txt”,”r”);
if(fp==NULL)
{
        printf(“\n file cannot be opened”);
        getch();
```

```
        exit(1);
}
fscanf(fp,"%d%f%c",&a,&b,&x);
printf("\n File contents are….");
printf("%d%f%c",a,b,x);
getch();
}
```

Write Mode

In write mode, file is opened for writing data to it.

Case 1: If given file is not present in the current/specified directory then compiler automatically creates new file with given name and opens the file returns first character address.

Case 2: If given file is present in the directory but it containing data then all the previous information which is present in the file is removed and returns the first character address.

- In case of write mode, always compiler returns first character address.
- In order to write information into file from 'c' program the following functions are used.

1. fputc()
2. fputs()
3. fprintf()

1. fputc(): This function is used to write single character into file.

Syntax: foid fputc(char ch, FILE *fp);

Example:

```
#include<stdio.h>
void main()
{
FILE *fp;
char ch='A',x;
fp=fopen("test.txt","w")'
fputc(ch,fp);
fclose(fp);
fp=fopen("test.txt","r");
x=fgetc(fp);
printf("\n The written character is %c",x);
fclose(fp);
getch();
}
```

2.fputs(): This function is used to write group of characters with in file.

Syntax: void fputs(char* strptr, FILE* fp);

Example:

```
#include<stdio.h>
void main()
{
char S[10]=”Hello”;
char T[10];
FILE *fp;
fp=fopen(“test.txt”,”w”);
fputs(S,fp);
fclose(fp);
fp=fopen(“test.txt”,”r”);
fgets(T,5,fp);
printf(“\n file contents are….\n”);
puts(T);
fclose(fp);
getch();
}
```

3.fprintf(): This function is used to write one record information into file. Record is nothing but collection different data types.

Syntax: void fprintf(fp,”formatstring”,addresslist);

Example:

```
#include<stdio.h>
void main()
{
int a,a1=10;
int b,b1=20.56;
char x,x1=’B’;
FILE *fp;
fp=fopen(“test.txt”,”w”);
fprintf(fp,”%d%f%c”,a1,b1,x1);
fclose(fp);
fp=fp[em(“test.txt”,”r”);
fscanf(fp,”%d%f%c”,&a,&b,&x);
printf(“\n file contents are…”);
printf(“%d%f%c”,a,b,x);
fclose(fp);
getch();
}
```

/*Program to copy contents of one file to another*/

```
#include<stdio.h>
void main()
{
FILE *fp1,*fp2;
char ch;
fp1=fopen(“abc.txt”,”r”);
if(fp1==NULL)
{
        printf(“\n file cannot opened”);
        getch();
        exit(1);
}
fp2=fopen(“test.txt”,”w”);
while(1)
{
        ch=fgetc(fp1);
        if(ch==EOF)
        break;
else
        fputc(ch,fp2);
}
fclose(fp1);
fclose(fp2);
fp2=fopen(“test.txt”,”r”);
while(1)
{
        ch=fgetc(fp2);
        if(ch==EOF)
        break;
else
        printf(“%c”,ch);
}
fclose(fp2);
getch();
}
```

Append mode:

- In append mode adds new information to already existing information that means append mode writes data from end of file.
- In case of write mode, if given file contains any previous information then that will be removed.

Example:

```
#include<stdio.h>
void main()
{
char ch,S[50]=”this is an example”;
FILE *fp;
fp=fopen(“abc.txt”,”a”);
fputs(S,fp);
fclose(fp);
```

```
fp=fopen(“abc.txt”,”r”);
while(1)
{
        ch=fgetc(fp);
        if(ch==EOF)
        break;
        printf(“%c”,ch);
}
fclose(fp);
getch();
}
```

Random access to files

Files can be accessed in two ways:

1. Sequential access
2. Random access

- Sequential access is, accessing information within the file from beginning position to ending position sequentially. In this case after opening file immediately we cannot access middle position or end position.

- In case of Random access any part of file is accessed. To perform Random access on files following functions are used.

1. ftell()
2. rewind()
3. fseek()

1. ftell(): This function returns the current position of the file pointer relative to the beginning of the file. Starting position of file is 0(zero).

Syntax: long ftell(FILE *);

Example:

```
#include<stdio.h>
void main()
{
char ch;
FILE *fp;
fp=fopen(“abc.txt”,”r”);
printf(“\n position=%ld”,ftell(fp));
ch=fgetc(fp);
printf(“\n position=%ld”,ftell(fp));
getch();
}
```

2.rewind(): The rewind() simply sets the file pointer to beginning position of the file.

Syntax: void rewind(FILE *)

Example:

```
#include<stdio.h>
void main()
{
int i,j=6;
char ch;
FILE *fp;
fp=fopen("abc.txt","r");
if(fp==NULL)
{
       printf("\n file cannot opened");
       getch();
       exit(1);
}
ch=fgetc(fp);
ch=fgetc(fp);
ch=fgetc(fp);
ch=fgetc(fp);
printf("\n position=%ld",ftell(fp));
rewind(fp);
printf("\n after rewinding position=%ld",ftell(fp));
fclose(fp);
}
```

3.fseek(): This function is used to move the file pointer to require position.

Syntax: int fseek(FILE *fp, long offset, int position);

If operation is successful then returns 0 otherwise returns 1.

- The offset specifies the number of positions to be moved from the location specified by position.
- The position can take one of the following three values.

Value	Meaning
0	Beginning position of file
1	Current position
2	End of file

The offset may be positive or negative, if it is positive then move forward. If it is negative move Backward.

Statement	**Meaning**
fseek(fp,0L,0);	Go to beginning(similar to rewind)
fseek(fp,0L,1);	Stay at current position
fseek(fp,0L,2);	Go to end of the file
fseek(fp,m,0);	Move to (m+1)th byte in the file
fseek(fp,+m,1);	Go forward by m bytes
fseek(fp,-m,1);	Go backward by m bytes from the current position
Fseek(fp,-m,2);	Go backward by m bytes from the end

r+ (or) a+ (or) w+

In this mode we can perform read, write and modifications to already existing data.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char ch;
FILE *fp;
clrscr();
fp=fopen("abc.txt","r+");
fseek(fp,0,2);
ch='B';
fputc(ch,fp);
rewind(fp);
while(1)
{
       ch=fgetc(fp);
       if(ch==EOF)
       break;
    else
       printf("%d",ch);
}
fseek(fp,3,0);
ch=fgetc(fp);
printf("%c",ch);
fseek(fp,4,1);
printf("%c",ch);
```

```
fseek(fp,-5,1);
ch=fgetc(fp);
fclose(fp);
getch();
}
```

Binary mode

In case of text mode, if we try to write information into the file, the compiler treats all the information as sequence of characters.

- In Binary mode two functions are used.

    1. **fread():** This function is used for reading an entire block from a given file.

    2. **fwrite():** This function is used for writing an entire structure block to given file.

Syntax of fread(): int fread(void* pin area, int elementsize, int count, FILE *);

- The first parameter is address of the structure to store information after reading it from file.

- The second parameter is size of structure.

- The third argument is the number of such structure that we want to read at one time.

- The last parameter is the pointer to the file we want to read.

Syntax of fwrite(): int fwrite(void* pin area, int elementsize, int count, FILE *);

- The first parameter is address of the structure to be written to the disk.

- The second parameter is the size of structure.

- The third argument is the number of such structure that we want to read one time.

- The last parameter is the pointer to the file we want to read.

feof(): The macro feof() is used for detecting the file pointer whether it is at the end of file or not. It returns non zero if the file pointer is at the end of file otherwise it return zero.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char ch;
fp=fopen(“abc.txt”,”r”);
clrscr();
while(!feof(fp))
{
        printf(“%c”,c);
        c=fgetc(fp);
}
fclose(fp);
}
```

ferror(): The ferror() is used to find out error when file read/write operations is carried out.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char ch;
fp=fopen(“abc.txt”,”r”);
if(fp==NULL)
{
        puts(“can not opened file”);
        exit(1);
}
while(1)
{
        printf(“\n Enter a character:”);
        scanf(“%c”,&ch);
        fputc(ch,fp);
 if(ferror(fp));
 {
        printf(“\n unable to write data”);
        fclose(fp);
        exit(1);
 }
if(ch==$)
break;
}
fclose(fp);
getch();
}
```

Output: Enter a character: A

Unable to write data

The program results above output because the file is opened in read mode. In read mode we can perform only read operation but in above program we are trying to perform write operation.