Chapter 1

User Defined Function

Introduction

Functions are a useful way of referring g to blocks of code, allowing the code to be called many times. Functions take some input information (the input parameters), do something with that information and then return a result (the return value). We are familiar with the predefined functions, such as gets(), puts(), getch( ), printf( ) and scanf( ). We call them Library Function or Built-In Functions. It is possible to define your own functions in C. Being able to do this allows powerful programs to be developed in a modular manner that makes the development and debugging stages significantly easier than would otherwise be the case.

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

Need For User-Defined Function

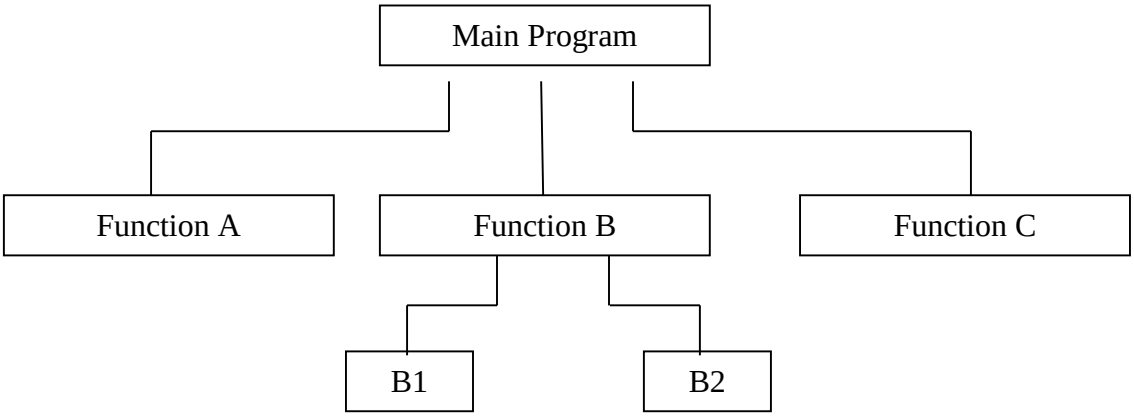
Every program must have main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program many become too large and complex and as a result the task of debugging, testing, and maintaining become difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit, these independently coded programs are called subprograms that are much easier to understand, debug, and test.

We already know that C support the use of library functions and use defined functions. The library functions are used to carry out a number of commonly used operations or calculations. The user-defined functions are written by the programmer to carry out various individual tasks.

There are many advantages in using functions in a program they are:

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.
3. It is easy to locate.
4. A function may be used by many other programs this means that a c programmer can build his own function.
5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program.

6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.



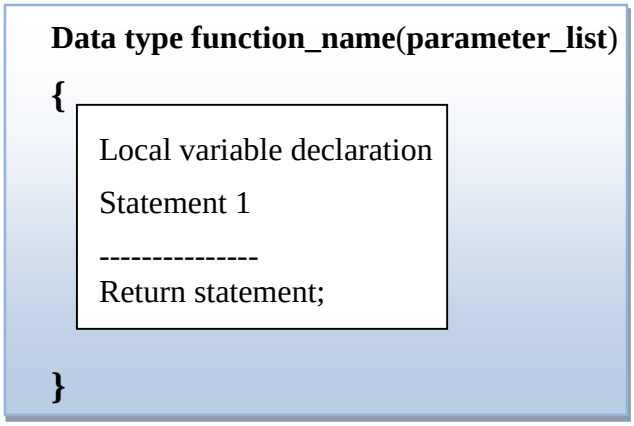
Top-Down modular Programming using function.

Functions are used in c for the following reasons:

1. Many programs require that a specific function is repeated many times instead of writing the function code as many times as it is required we can write it as a single function and access the same function again and again as many times as it is required.
2. We can avoid writing redundant program code of some instructions again and again.
3. Programs with using functions are compact & easy to understand.
4. Testing and correcting errors is easy because errors are localized and corrected.
5. We can understand the flow of program, and its code easily while using the functions.
6. A single function written in a program can also be used in other programs also.

Defining Functions

A function is defined as follows:



Type

Functions return a value. The type of this value (int, float, char etc) has to be specified. All of the standard C types met so far may be used as a function type. Sometimes a return value is not required, for example if the function prints an error message and then returns, in which case a return type of void should be used.

function name

This is a label that is used to refer to the function from other code blocks. The rules for valid names are the same as those for variables.

Parameter list

Values are passed to a function via its parameter list. This is a comma separated list of the values that the function takes. Each item in the list has a type specification. The parameter list can be blank, i.e. the function takes no parameters, in this case the keyword void should be used instead of a list (as is the case with "int main(void)". The names used for the parameters can be different from those used in the block of code that calls the function.

Code block

The code block contains the operations that constitute the function. The parameters declared in the parameter list can be used as standard variables within this block. Temporary variables may also be declared for use within the block. Unless the function has a return type of void then the function needs to return a value of the correct type. This is done via the return command (see below).

A simple example

```
#include <stdio.h>
#include <stdlib.h>
float product(float x, float y)
{
        float result; // a local variable
        result = x*y;
        return result; // returning the value
}
int main(void)
{
        float a;
        float b;
        float answer;
        printf("Please input two values to be multiplied:");
        scanf("%f %f",&a,&b);
        // call the function product
        answer=product(a,b); // calling the function
        printf("The product of %f and %f is %f\n",a,b,answer);
        system("pause");
        return 0;
}
```

Notes

1. This example is rather too simple to be useful, but does illustrate how functions are defined and used.
2. The function product is defined here. It has two input parameters and returns a value of type float.
3. result is a variable declared within the product code block. It can only be used in this block.
4. The function returns the value stored in result (which is of type float).
5. The values of a and b are passed to the function product, the result is stored in the variable answer. Notice that different parameter names are used in the function and main block.

* Simple function - with prototype

```
#include <stdio.h>
```

```
#include <stdlib.h>
float product(float x, float y);
int main(void)
{
        float a;
        float b;
        float answer;
        // prompt user for values
        printf("Please input two values to be multiplied:");
        scanf("%f %f",&a,&b);
        // call the function product
        answer=product(a,b);
        printf("The product of %f and %f is %f\n",a,b,answer);
        system("pause");
        return 0;
}
float product(float x, float y)
// function to return the product of two numbers
{
        float result; // a local variable
        result = x*y;
        return result; // returning the value
}
```

Passing Values between Functions

The functions that we have used so far haven't been very flexible. We call them and they do what they are designed to do. Like our mechanic who always services the motorbike in exactly the same way, we haven't been able to influence the functions in the way they carry out their tasks. It would be nice to have a little more control over what functions do, in the same way it would be nice to be able to tell the mechanic, "Also change the engine oil, I am going for an outing". In short, now we want to communicate between the 'calling' and the 'called' functions.

The mechanism used to convey information to the function is the 'argument'. You have unknowingly used the arguments in the **printf( )** and **scanf( )** functions; the format string and the list of variables used inside the parentheses in these functions are arguments. The arguments are sometimes also called 'parameters'.

Consider the following program. In this program, in **main( )** we receive the values of **a**, **b** and **c** through the keyboard and then output the sum of **a**, **b** and **c**. However, the calculation of sum is done in a different function called **calsum( )**. If sum is to be calculated in **calsum( )** and values of **a**, **b** and **c** are received in **main( )**, then we must pass on these values to **calsum( )**, and once **calsum( )** calculates the sum we must return it from **calsum( )** back to **main( )**.

```
/* Sending and receiving values between functions */
main( )
{
        int a, b, c, sum ;
        printf ( "\nEnter any three numbers " ) ;
        scanf ( "%d %d %d", &a, &b, &c ) ;
        sum = calsum ( a, b, c ) ;
        printf ( "\nSum = %d", sum ) ;
}
calsum ( x, y, z )
int x, y, z ;
{
        int d ;
        d = x + y + z ;
        return ( d ) ;
}
```

Out Put:-

```
Enter any three numbers 10 20 30
Sum = 60
```

Return Types

As pointed out earlier, a function may or may not send back any value to the calling function. If it does, it is done through the return statement. While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most.

The return statement can take one of the following forms.

```
return;
or
return(expression);
```

the first, the 'plain' return does not return any value, it acts much as the closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function. An example of the use of a simple return is as follows.

```
If(error)
Return;
```

The second form of return with an expression returns the value of the expression.

Ex:-

```
Int mul(int x, int y)
{
    Int p;
    P=x*y;
    Return(p);
}
```

Returns the value of p which is the product of the values of x and y. the last two statements can be combined into one statement as follows.

```
Return(x*y);
```

A function may have more than one return statements. This situation arises when the value returned is based on certain conditions ex

```
If(x<=0)
    Return(0);
Else
    Return(1);
```

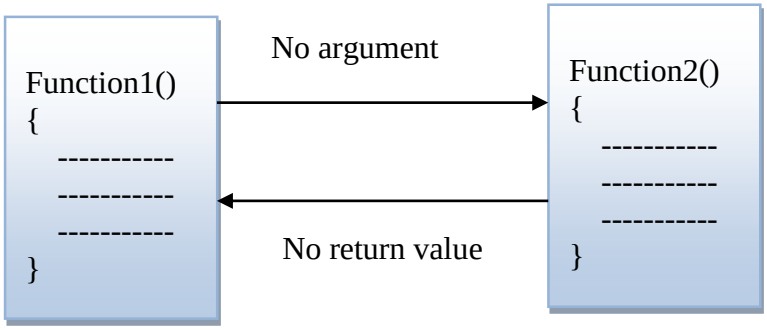
Types of functions:

A function may belong to any one of the following categories:

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions no arguments and with return values.

Functions no arguments and no return values:

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. Similarly when it does not return value the calling function does not receive any data from the called function. A function that does not return any value cannot be used in an expression it can be used only as independent statement.



Let us consider the following program

```
/* Program to illustrate a function with no argument and no return values*/
/* #includes go here followed by the three prototypes */
void statement1();
void statement2();
void starline();
void main()
{
        staetement1();
        starline();
        statement2();
        starline();
}
/*function to print a message*/
void statement1()
{
        printf("\n Sample subprogram output");
}
void statement2()
```

```
{
        printf("\n Sample subprogram output two");
}
void starline()
{
        int a;
        for (a=1;a<60;a++)
        printf("%c",'*');
        printf("\n");
}
```

Functions with arguments but no return values:

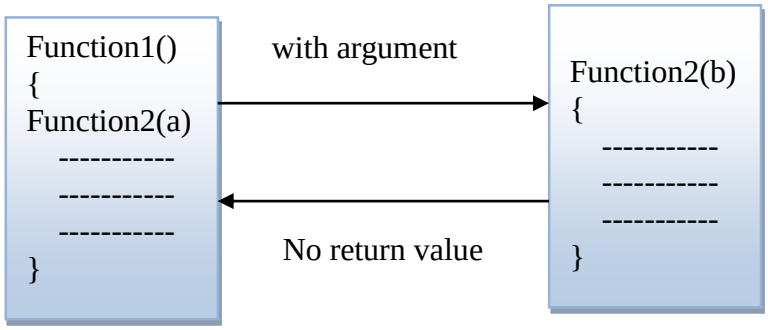
The nature of data communication between the calling function and the arguments to the called function and the called function does not return any values to the calling function.

The no of formal arguments and actual arguments must be matching to each other suppose if actual arguments are more than the formal arguments, the extra actual arguments are discarded. If the number of actual arguments is less than the formal arguments then the unmatched formal arguments are initialized to some garbage values. In both cases no error message will be generated.

The formal arguments may be valid variable names, the actual arguments may be variable names expressions or constants. The values used in actual arguments must be assigned values before the function call is made.

When a function call is made only a copy of the values actual argument is passed to the called function. What occurs inside the functions will have no effect on the variables used in the actual argument list.

Both the arguments actual and formal should match in number type and order. The values of actual arguments are assigned to formal arguments on a one to one basis starting with the first argument as shown below:



```
main()
{
        function1(a1,a2,a3……an)
}
function1(f1,f2,f3….fn);
{
        function body;
}
```

here a1,a2,a3 are actual arguments and f1,f2,f3 are formal arguments.

```
/*Program to find the largest of two numbers using function*/
/* #include */
void largest(int, int);
main()
{
        int a,b;
        printf("Enter the two numbers");
        scanf("%d%d",&a,&b);
        largest(a,b)
}
/*Function to find the largest of two numbers*/
void largest(int a, int b)
{
        if(a>b)
        printf("Largest element=%d",a);
        else
        printf("Largest element=%d",b);
}
```

In the above program we could make the calling function to read the data from the terminal and pass it on to the called function. But function foes not return any value. So the return type of the function is void.

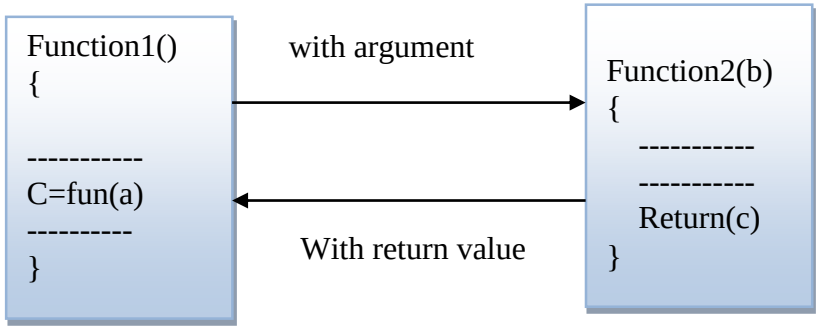
Functions with arguments and with return values:

The function of the type Arguments with return values will send arguments from the calling function to the called function and expects the result to be returned back from the called function back to the calling function.

In the above type of function the following steps are carried out:

1. The function call transfers the controls along with copies of the values of the actual arguments of the particular function where the formal arguments are creates and assigned memory space and are given the values of the actual arguments.
2. The called function is executed line by line in normal fashion until the return statement is encountered. The return value is passed back to the function call is called function.

3. The calling statement is executed normally and return value is thus assigned to the calling function.



Return value data type of function:

A C function returns a value of type int as the default data type when no other type is specified explicitly. For example if function does all the calculations by using float values and if the return statement such as return (sum); returns only the integer part of the sum. This is since we have not specified any return type for the sum. There is the necessity in some cases it is important to receive float or character or double data type. To enable a calling function to receive a non-integer value from a called function we can do the two things:

1. The explicit type specifier corresponding to the data type required must be mentioned in the function header. The general form of the function definition is

Type_specifier function_name(Type_specifier1 Argument1, Type_specifier2 Argument2, …)

{

function statement;

}

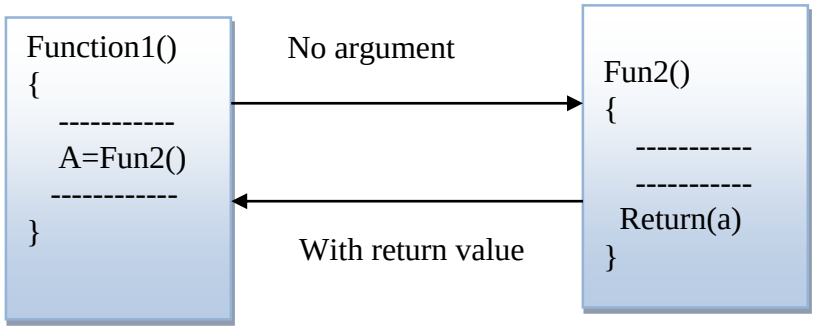
The type specifier tells the compiler, the type of data the function is to return.

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data the function is actually returning. The program given below illustrates the transfer of a floating-point value between functions done in a multiple function program.

```
float add(float,float);
double sub(double,double);
main()
{
        float x,y;
        x=12.345;
        y=9.82;
        printf("%f\n" add(x,y));
        printf("%lf\n"sub(x,y);
}
float add(float a, float b)
{       return(a+b);
}
double sub(double p, double q)
{       return(p-q);
}
```

Function no arguments and with return types:

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. But when it does return value the calling function receives the data from the called function and we store it in a variable. Here the variable sum receives the summation of two numbers returned from the function add().




```
/* Program to illustrate a function with no argument and return values*/
/* #includes go here */
```

```
int add();
void main()
{
int sum;
sum=add();
printf("Sum of two nos:%d",sum);
}
int add()
{
int a,b,sum;
printf("Enter two nos:");
scanf("%d%d",&a,&b);
sum=a+b;
return(sum);
}
```

Nesting of functions:

C permits nesting of two functions freely. There is no limit how deeply functions can be nested. Suppose a function a can call function b and function b can call function c and so on. Consider the following program:

```
Void add(int, int);
Void sub(int, int);
void main()
{
        int a,b,c;
        scanf("%d%d",&a,&b);
        add(a,b);
}
Void add(int x, int y)
{
        int z;
        z=x+y;
        printf("\nAdd=%d",z);
```

```
        sub(x,y);
}
Void sub(int p, int q)
{
        int z;
        z=p-q;
        Printf("\nSub=%d",z);
}

}
```

the above program calculates the ratio a/b-c; and prints the result. We have the following three functions: main(), ratio() and difference() main reads the value of a,b,c and calls the function ratio to calculate the value a/b-c) this ratio cannot be evaluated if(b-c) is zero. Therefore ratio calls another function difference to test whether the difference(b-c) is zero or not.

UNIT II

Storage Classes

We have already said all that needs to be said about constants, but we are not finished with variables. To fully define a variable one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a 'storage class.

We have not mentioned storage classes yet, though we have written several programs in C. We were able to get away with this because storage classes have defaults. If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Thus, variables have certain default storage classes.

From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept— Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

A variable's storage class tells us:

(a) Where the variable would be stored.
(b) What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value is garbage).
(c) What is the scope of the variable; i.e. in which functions the value of the variable would be available.
(d) What is the life of the variable; i.e. how long would the variable exist.

There are four storage classes in C:

(a) Automatic storage class
(b) Register storage class
(c) Static storage class
(d) External storage class

Automatic Storage Class

Automatic storage class are variables which are declared inside a function. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. These variables are private or local to the function in which they are declared.

Storage	Memory.
initial value	Garbage value.
Scope	Local to the block in which the variable is defined.

Life	Till the control remains within the block in which the variable is defined.

Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

```
main( )
{
auto int i, j ;
printf ( "\n%d %d", i, j ) ;
}
```

Output:-

1211 221

where, 1211 and 221 are garbage values of i and j. When you run this program you may get different values, since garbage values

Scope and life of an automatic variable is illustrated in the following program.

```
main( )
{
auto int i = 1 ;
{
{
{
printf ( "\n%d ", i ) ;
}
printf ( "%d ", i ) ;
}
printf ( "%d", i ) ;
}
}
```

The output of the above program is:

1 1 1

```
main( )
{
auto int i = 1 ;
{
auto int i = 2 ;
{
auto int i = 3 ;
printf ( "\n%d ", i ) ;
}
printf ( "%d ", i ) ;
}
printf ( "%d", i ) ;
}
```

Understand the concept of life and scope of an automatic storage class variable thoroughly before proceeding with the next storage class.

Register Storage Class:-

Register storage class is a variable which is stored in CPU register instead of memory. The access speed in register is much faster than memory which enables the faster execution of programs.

The microprocessor has 16-bit registers then they cannot hold a float value or a double value, which require 4 and 8 bytes respectively. However, if you use the register storage class for a float or a double variable you won't get any error messages. All that would happen is the compiler would treat the variables to be of auto storage class.

The features of a variable defined to be of register storage class are as under:

Storage	CPU registers.
Default initial value	Garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

```
main( )
{
        register int i ;
        for ( i = 1 ; i <= 10 ; i++ )
        printf ( "\n%d", i ) ;
}
```

Static Storage Class:-

Like auto variables, static variables are also local to the block in which they are declared. The difference between them is that static variables don't disappear when the function is no longer active. Their values persist. If the control comes back to the same function again the static variables have the same values they had last time around.

The features of a variable defined to have a static storage class are as under:

Storage	Memory.
Initial value	Zero.
Scope	Local to the block in which variable is declared
Life	Till the control remain within the block.

```
main( )
{
        increment( ) ;
        increment( ) ;
        increment( ) ;
}
increment( ) { static int i = 1 ;
printf ( "%d\n", i ) ;
i = i + 1 ;
}

main( )
{
        increment( ) ;
        increment( ) ;
        increment( ) ;
}
increment( )
{
        auto int i = 1 ;
        printf ( "%d\n", i ) ;
         i = i + 1 ;
}
```

In the above example, when variable i is auto, each time increment( ) is called it is re-initialized to one. When the function terminates, i vanishes and its new value of 2 is lost. The result: no matter how many times we call increment( ), i is initialized to 1 every time.

On the other hand, if i is static, it is initialized to 1 only once. It is never initialized again. During the first call to increment( ), i is incremented to 2. Because i is static, this value persists. The next time increment( ) is called, i is not re-initialized to 1; on the contrary its old value 2 is still available. This current value of i (i.e. 2) gets printed and then i = i + 1 adds 1 to i to get a value of 3. When increment( ) is called the third time, the current value of i (i.e. 3) gets printed and once again i is incremented. In short, if the storage class is static then the statement static int i = 1 is executed only once, irrespective of how many times the same function is called.

External Storage Class:-

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

The features of a variable whose storage class has been defined as external are as follows:

Storage	Memory.
Default initial value	Zero.
Scope	Global.
Life	As long as the program's execution doesn't come to an end.

```
int i ;
main( )
{
        printf ( "\ni = %d", i ) ;
        increment( ) ;
        increment( ) ;
        decrement( ) ;
        decrement( ) ;
}
increment( )
{
        i = i + 1 ;
        printf ( "\non incrementing i = %d", i ) ;
}
decrement( )
{
        i = i - 1 ;
        printf ( "\non decrementing i = %d", i ) ;
}
```

output

```
i = 0
on incrementing i = 1
on incrementing i = 2
on decrementing i = 1
on decrementing i = 0
```

Call by value

In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

In call by value method, we can not modify the value of the actual parameter by the formal parameter.

In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
}
void swap (int a, int b)
{
   int temp;
   temp = a;
   a=b;
   b=temp;
   printf("After swapping values in function a = %d, b = %d\n",a,b);
 }
```

Call by reference

In call by reference, the address of the variable is passed into the function call as the actual parameter.

The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
 }
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d, b = %d\n",*a,*b);
 }
```

Recursion

In C it is possible for a function to call itself, a process known as recursion. A function that calls itself is said to be recursive. A recursive function requires a call to itself and a test to see if some stop condition has been met bringing to an end the recursion.

Consider for example the calculation of a factorial.

```
int factorial (int n)
{
int i;
int fact=1;
if (n>=1)
for (i=1;i<=n;i++) fact=fact*i;
return fact;
}
```

The use of unsigned int in order to ensure that only positive integers are used. Another way of calculating a factorial comes out of realizing that: i.e. n factorial is n times (n-1) factorial.

```
main( )
{
        int a, fact ;
        printf ( "\nEnter any number " ) ;
        scanf ( "%d", &a ) ;
        fact = rec ( a ) ;
        printf ( "Factorial value = %d", fact ) ;
}
rec ( int x )
{
        int f ;
        if ( x == 1 )
        return ( 1 ) ;
        else
        f = x * rec ( x - 1 ) ;
        return ( f ) ;
}
```

Notest

1. This line terminates the recursion process.
2. This is the recursive call to factorial
3. This function has two return statements. This is allowed in C

Recursion is not just for moderately obscure mathematical functions. It turns out that many problems when considered properly can be written in terms of recursion. The reading/writing of lists from/to a file can readily be implemented in a recursive manner. Another example is the Towers of Hanoi problem that was very popular as a toy in Victorian times.

UNIT III

Strings in C

What Are String

A string is a sequence of character that is treated as a single data item. We have used string in a number of examples in the past.

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings. Many languages internally treat strings as character arrays, Character arrays or strings are used by programming languages to manipulate text such as words and sentences.

A string constant is a one-dimensional array of characters terminated by a null ('\0'). For example,

```
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies one byte of memory and the last character is always '\0'. What character is this? It looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special. '\0' is called null character. Note that '\0' and '0' are not same. ASCII value of '\0' is 0, whereas ASCII value of '0' is 48. Figure 9.1 shows the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations.

The terminating null ('\0') is important, because it is the only way the functions that work with a string can know where the string ends. For example, the string used above can also be initialized as,

```
char name[ ] = "HAESLER" ;
```

In this declaration '\0' is not necessary. C inserts the null character automatically.

H	A	E	S	L	E	R	'\0'
65525	65524	65523	65522	65521	65520	65519	65518

Declaring And Initializing String Variables

C does not support strings as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is

Syntax:-

```
Char string_Name[size];
```

The size determines the number of characters in the strine_name. some examples are.

Char city[10];
Char name[20];

When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms.

Char city[9]="NEW YORK";
Char city[9]={'N','E','W',' ','Y','O','R','K','\0'};

The reason that city had to be 9 elements long is that the string NEW YORK contains characters and one element space is provided for the null terminator. Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

Char string[]={'G','O','O','D','\0'};

We can also declare the size much large than the string size in the initialize. That is, the statement.

Char str[10]="GOOD";

In this case the computer creates a character array of size 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL.

G	O	O	D	\0	\0	\0	\0	\0	\0

Declaration is illegal

Char xtr1[2]="GOOD";

Standard Library String Function.

The c library support a large number of string handling functions that can be used to carry out many of the string manipulation discussed so far.

strcat function:-

The strcat function join two string together. It takes the following form

strcat(string1,string2)

Sstring1 and string 2 are character arrays, when the function strcat is executed, string2 is appended to string1. It does so by removing the null character at the end of string1 and placing string2 from there. The string at string2 remains unchanged.

Ex:- /*Program Study Charcter string Display strcat Function*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        char source[]="Bhosle";
        char target[20]="Hello";
        clrscr();
        strcat(target,source);
        printf("\n\t\t---------OUTPUT----------\n");
        printf("\n\n \t\tTarget  String:=%s",target);
        getch();
}
```

Strlen

This function count numbers of character present in the string it means. it return length of the string consider following example to understand the use of strlen function.

Integer variable=strlen(string);
Ex:- strlen function

Calculate the length of the string which return integer value. That value is assign to integer variable.

Ex:-/* write a program to demostrate the use of
strlen() and strupr(). */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        char a[20];
        int n;
        clrscr();
        printf(" \n\n\t Enter the string : ");
        scanf("%s",&a);
        n=strlen(a);
        printf("\n\n\tThe length of given string : %d ",n);
```

```
        n=strupr(a);
        printf(" \n\n\t String in UPPER case  : %s",a);
        getch();
}
```

Strcpy

This string library function is used to copy one string into another string.

Syntax:-

Strcpy(source string, target string)

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        char a[6]="Bhosle";
        char b[6];
        clrscr();
        printf("string= %s",a);
        strcpy(b,a);
        printf("string %s",b);
        getch();
}
```

Strcmp

This function compare to different thing & finds out whether given string are some or different the two strings are compared char offend by character until there is mismatch or end of one of string reached it is two string are identical (same) then the function return a 0 value but if they are not same then it returns numeric difference between the ASCII value of first non matching pains of character.

```
/*Program Study Charcter string strcpy & strcmp Function
        Display Given String is Palendrome or Not*/

        #include<stdio.h>
        #include<conio.h>
        #include<string.h>
        void main()
        {
                char ab[10],ar[10];
                clrscr();
                printf("\n\t\t---------OUTPUT----------\n");

                printf("\n\t\tEnter The String:=");
                scanf("%s",ab);

                strcpy(ar,ab);
```

```
        if(strcmp(ab,strrev(ar))==0)
        {
                printf("\n \t\tThis is Palendrome String",ab);
        }
        else
        {
                printf("\n\t\tThis is Not Palendrome string",ab);
        }

        getch();
}
```

Strupper

This function is used to convert the string lower case letter into upper case letter. If given string is already in the upper case letter it remains as it is.

Ex:

```
Void main()
{
        Char name[15]=”smita”;
        Strupper(name);
        Printf(“name= %s”,name);
        Getch();}
}
```

Strlwr

This function is used to convert the string upper case letter into lower case letter. If given string is already in the lower case letter it remains as it is.

Ex:

```
Void main()
{
        Char name[15]=”smita”;
        Strlwr(name);
        Printf(“name= %s”,name);
        Getch();}
}
```

Strrev

This function reverse the given string and stores it at the same place.

```
Main()
{
         Static char college[6]=”COCSIT”;
        Strrev(college);
        Printf(“Name of the College is %s”,college);
```

```
    Getch();
}
```

Strings and pointers

String is a data type that stores the sequence of characters in an array. A string in C always ends with a null character (\0), which indicates the termination of the string. Pointer to string in C can be used to point to the starting address of the array, the first character in the array. These pointers can be dereferenced using the **asterisk** * **operator** to identify the character stored at the location. 2D arrays and pointer variables both can be used to store multiple strings.

```
#include <stdio.h>

int main(void) {
  char name[] = "Harry Potter";

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o

  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);     // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

Array of Strings

An Array is the simplest Data Structure in C that stores homogeneous data in contiguous memory locations. If we want to create an Array, we declare the Data type and give elements into it:

```
#include<stdio.h>
int main()
{
    char str[8];
    printf("Enter a String: ");
    scanf("%s", &str);
    printf("%s", str);
}
```

Chapter 4
Pointers

Introduction To Pointers

A pointer is a derived data type in C. it is built from one of the fundamental data types available in C. pointer contain memory addresses as their values. Since these memory addressed are the locations in the computer memory where program instructions and data are stored, pointer can be used to access and manipulate data stored in the memory.
It has added power and flexibility to the language.
Pointers are used frequently in C, as they offer a number of benefits to the programmers.

1. Pointers are more efficient in handling arrays and data tables.
2. Pointer can be used to return multiple values from a function or function arguments.
3. Pointer permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character string results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked list.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Declaring pointer variables:

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointer before we use them.
The declaration of a pointer variable takes the following form.

```
Syntax:
        Data_type *ptr_name;
```

Rules

1. The asterisk (*) tells that the variable ptr_name is a pointer variable
2. Ptr_name needs a memory location.
3. Ptr_name point to a variable of type data_type.

Ex:

```
Int  *p;
```

Declare the variable p as pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

```
Float *x;
```

Declares x as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory location for the pointer variables p and x. since the memory locations have not been assigned any values. These locations may contain some unknown values in them and therefore they point to unknown locations.
Pointer variable are declared similarly as normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the printer variable name.

```
int* p;
int *p;
int * p;
```

//WAP to demonstrate the pointer

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int *i;
        int j=70;
        i=&j;
        clrscr();
        printf("The address of j is stored in pointer variable i %u",i);
        printf("\ndisplay the value of j using pointer variable i %u",*i);
        getch();
}
```

Initialization of pointer variables:

The process of assigning the address of a variable to a pointer variable is known as initialization. As pointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addressed. They may not be valid addressed or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce result. It is therefore important to initialize pointer variable carefully before they are used in the program.

-Once a pointer variable has been declared we can use the assignment operator to initialize the variable.

```
int q;
int *p;
p=&q;
int *p=&q;
```

-The only requirement here is that the variable q must be declared before the initialization takes place.
-We must ensure that the pointer variables always point to the corresponding type of data.
Ex

```
float a,b;
int x,*p;
p=&a;  /* wrong */
b=*p;
```

will result incorrect output as we are trying to assign the address of a float variable to an integer pointer. When we declare a pointer to be of int type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors. Care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one time.

```
int x, *p=&x;
```

is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. and also remember that the target variable x is declared first.

```
Int *p=&x, x;
```

Is not valid. We could also define a pointer variable with an initial value of NULL or 0. Ex the following statement are valid.

```
int *p=NULL;
int *p=0;
```

with the exception of NULL and 0, no other constant value can be assigned to a pointer variable. Ex-the following is wrong

```
int *p=5360;
```

Accessing a Variable through its pointer:

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is done by using another unary operator * (asterisk), usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator.

Consider the following statement.

```
int quantity, *p, n;
quantity=160;
p=&quantity;
n=*p;
```

The first line declares quantity and n as integer variables and p as a pointer variable pointing to an integer. The second line assign the value 160 to quantity and the third line assign the address of quantity to the variable p. the fourth line contains the indirection operator (*). When the operator * is placed before a pointer variable in an expression. The pointer returns the value of the variable of which the pointer value is the address. In this case *p returns the value of the variable quantity, because p is the address of quantity. The * can be remembered as value at address. Thus the value of n would be 160.

```
p=&quantity;
n=*p;
```

are equivalent to

```
n=*&quantity;
```

which is turn is equivalent to

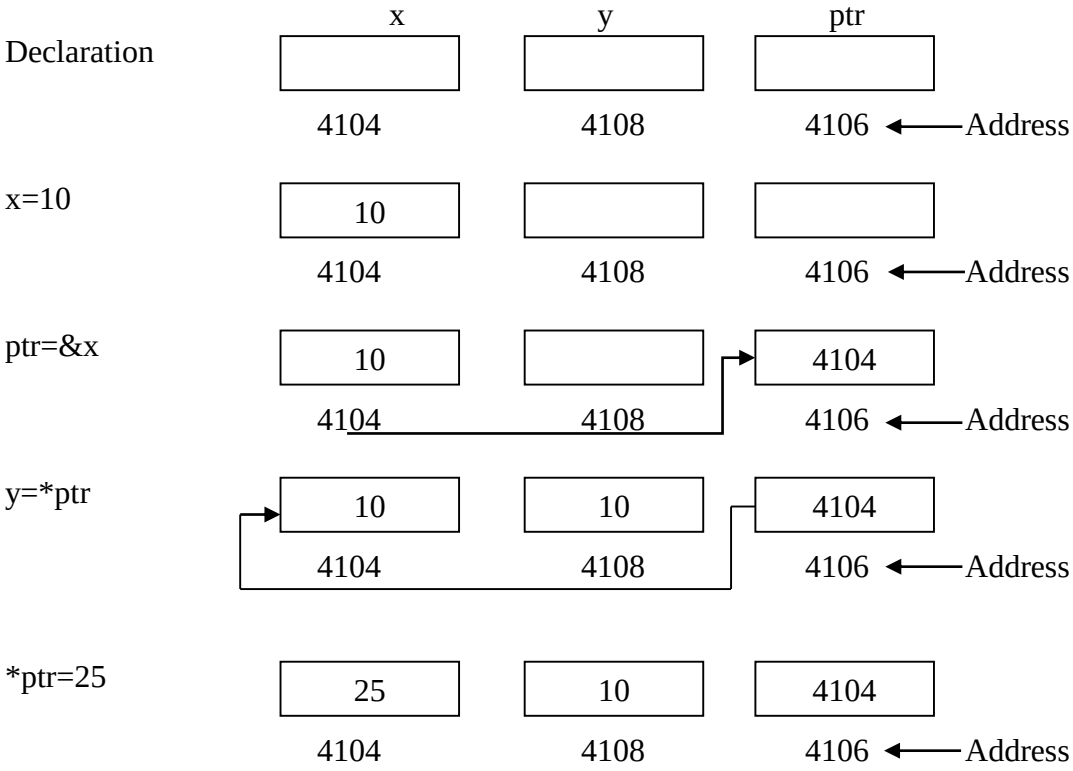
```
n=quantity;
```

```
void main()
{
```

```
        int x, y;
        int *ptr;
        x=10;
        ptr=&x;
        y=*ptr;
        printf("value of x is %d\n\n",x);
        printf("%d is stored at address %u\n",x, &x);
        printf("%d is stored at address %u \n",*&x,&x);
        printf("%d is stored at address %u\n",*ptr,ptr);
        printf("%d is stored at address %u\n",ptr,&ptr);
        printf("%d is stored at address %u\n",y,&y);
        *ptr=25;
        printf("\nNow x=%d\n",x);
        getch();
    }
```

The statement p=&x assign the address of x to p
y=*p; assign the value pointed to buy the pointer p to y.
*p=25;

This statement puts the value of 25 at the memory location whose address is the value of ptr. We know that the value of ptr is the address of x and therefore the old value of x is replaced by 25.



Pointer to pointer:

Pointer is known as a variable containing an address of another variable. The pointer variable also have an address. The pointer variable containing address of another pointer variable is called as pointer to pointer.

Ex:

```
void main()
{
        int a=2, *p, **q;
        p=&a;
        q=&p;
        printf("Value of a is %d address of a =%u",a, a, &a);
        printf("The &a);
        printf("The value of *p of a=%d address of a is %u",*p,p);
        printf("**q to value of a =%d address of a =%u",**q,*q);
        getch();
}
```

In the above program variable p is declared as a pointer. There variable q is declared as pointer to pointer. They are declared as '*p' & '**q' the address of variable a is assigned to '*p' the address of *p is assign to q and display address & value of variable a.

Pointer Expression

Like other variables, pointer variable can be used in expressions. Ex if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1 * *p2;
sum=sum + *p1;
z=5*  - *p2/ *p1;
*p2=*p2 + 10;
```

There is blank space between / and * in the item 3 above. The following is wrong.

```
Z=5* - *p2 /*p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. P1+4, p2-2 and p1-p2 are all allowed. If p1 and p2 are both pointers to the same array, then p2-p1 gives the number of elements between p1 and p2.

We may also use short-hand operators with the pointer.

Ex-

```
p++;
--p;
```

In addition to arithmetic operators discussed above, pointer can also be compared using the relational operators. The expression such as p1>p2, p1==p2, and p1 !=p2 are allowed.

\\Write a program to different types of operators

```
Void main()
{
```

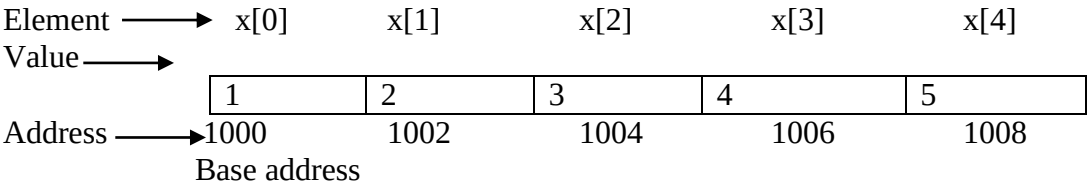
```
int a, b, *p1, *p2, x, y, z;
a=12;
b=4;
p1=&a;
p2=&b;
x=*p1 * *p2-6;
y=4* - *p2 / *p1 + 10;
printf("address of a= %u\n",p1);
printf("Address of b=%u\n',p2);
printf("\n");
printf("a=%d, b=%d\n",a,b);
printf("x=%d, y=%d\n",x,y);
*p2 = *p2 + 3;
*p1= *p2 – 5;
Z= *p1 * *p2 – 6;
printf("\na= %d, b = %d",a,b);
printf("z=%d\n",z);
getch();
}
```

Arrays And Pointer:

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The computer also defines the array name as a constant pointer to the first element, suppose we declare an array x as follows.

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

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows.



The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored.

```
x=&x[0]=1000
```

If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the following assignment.

```
p=x;
p=&x[0];
```

we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown as

```
p=&x[0](=1000)
p+1=&x[1](=1002)
p+2=&x[2](=1004)
```

p+3=&x[3](=1006)
p+4=&x[4](=1008)

You may notice that the address of an element is calculated using its index and the scale factor of the data type.

```
Void main()
{
        int *p, sum, i;
        int x[5]={5,9,6,3,7};
        i=0; p=x;
        printf("Element value address\n\n");
        while(i<5)
        {
                printf("x[%d] %d %u\n",i,*p, p);
                sum=sum+*p;
                i++; p++;
        }
        printf("\n Sum= %d\n",sum);
        printf("\n &x[0] = %u\n",&x[0]);
        printf("\n p = %u\n",p);
        getch();
}
```

Function And Pointer (call by Reference)

call by reference, the address of the variable is passed into the function call as the actual parameter. The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed. In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(&x);//passing reference in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Pointer To Function

When we declare any function the compiler can allocate the memory space to the particular function. Some time we need to store the address of function. Then we should declare the pointer to function. The pointer to function are used for storing the address of function and we can call the function using pointer to function.

```
Void main()
{
type (*pointer-name)(parameter);

type functionname();

pointer variable = function name;

(pointer variable)();

}

Type function name()

{

}
```

Ex:

```
Void main()
{
Void show();
Void (*p)();
P=show;
}

Void show()
{
-----
}
```

DYNAMIC MEMORY ALLOCATION IN C:

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

The process of allocating memory during program execution is called dynamic memory allocation. 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() FUNCTION IN C:

malloc () function is used to allocate space in memory during the execution of the program.
malloc () does not initialize the memory allocated during execution. It carries garbage value.
malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

Syntax of malloc()

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

Example

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

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
 int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = malloc( 20 * sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
      strcpy( mem_allocation,"fresh2refresh.com");
    }
    printf("Dynamically allocated memory content : " \
```

```
        "%s\n", mem_allocation );
    free(mem_allocation);
}
```

CALLOC() FUNCTION IN C:

The calloc () function is also used for allocate the memory at the run time it is same like malloc () function. Only difference is that wen we allocate memory using calloc function the variable contents initially zero value.

Syntax of calloc()

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

Example:

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

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = calloc( 20, sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
       strcpy( mem_allocation,"fresh2refresh.com");
    }
       printf("Dynamically allocated memory content   : " \
             "%s\n", mem_allocation );
       free(mem_allocation);
}
```

REALLOC() FUNCTION IN C:

realloc () 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.

Syntax of calloc()

```
ptr = (castType*)realloc(p, n, size);
```

Example:

```
ptr = (float*) realloc(p, 25, sizeof(float));
```

FREE() FUNCTION IN C:

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

Syntax of free()

```
free(ptr);
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,"fresh2refresh.com");
   }
   printf("Dynamically allocated memory content  : " \
         "%s\n", mem_allocation );
   mem_allocation=realloc(mem_allocation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
      strcpy( mem_allocation,"space is extended upto " \
                        "100 characters");
   }
   printf("Resized memory : %s\n", mem_allocation );
   free(mem_allocation);
}
```

Command Line Argument

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

```
#include <stdio.h>

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

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

Chapter 5
Structures And Union

Introduction

We know that the arrays can be used to represent a group of data items that belong to the same type, such as int or float. However if we want to represent a collection of data items of different types using a single name, then we cannot use on array.

'C' supports a constructed data type known as structure, which is a method for handling a group of logically related data items. It can be used to represent a set of attributes, such as student_name, roll_no, etc. the concept of structures is analogous to that of a 'record' in many other languages.

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

Structure Definitions

A structure definition creates a format that may be used to declare the structure variable. Let us consider an example that a process of structure definition and the creation of structure variable.

Syntax:-

```
struct tag_name
{
        Data_type member1;
        Data_type member2;
        ----------------------
        ----------------------
};
```

Consider a book database consisting a book_name, author, page_no, and price. We can define a structure to hold this information as.

```
struct book_bank
{
        char title[10];
        char author[15];
        int page;
        float price;
}
```

The keywords struct declares the a structures to hold the details of four fields namely title, author, pages, and price. These fields are called as structure elements or members. Each member may be belong to a different type of data book_bank is the name of structure and is called as structure tag. The tag name may be used subsequently to declares the variables that have the tag's structure.

Note that the above declaration has not declared any variables. It simply describes a format called template to represent a information as shown below.

```
struct book_name
{
```

```
        title array of 20 characters.
        author aray of 15 character.
        pages integer.
        price float.
}
```

We can declare the structure variable using the tag name anywhere in the program.
In defining a structure you may note the following syntax.
-The template is terminated with a semicolon (;)
-While the entire declaration is considered as a statement, each member is declared independently
-for its name and type in a separate statement inside the template.
-The tag name can be used to declare structure variable of its type, later in the program.

Declaring structure variable:

After defining a structure format we can declare variable of that type. A structure variable declaration is similar to the declaration of variable of any other data types. It includes the following elements.

1. The keyword struct.
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

```
Struct book_bank
{
        char title[10];
        char author[15];
        int pages;
        float price;
}book1,book2,boo3;
```

Or

```
Struct book_bank book1,book2,book3;
```

Declares book1, book2 and book3 as variables of type struct book_name.
Valid. The use of tag name is optional.

```
Struct
{
        ---------
        ---------
}book1,book2,book3;
```

The book1, book2, book3 is the structure variable representing three books, but does not includes the tage name for later use in declarations. Normally, structure definition appears at the begging of the programming file, before any variables or functions are defined. They may also appear before the main, along with macro definition, such as #define. In such cases, the definition is global and can be used by other functions as well.

Structure Initialization:

Like any other data type, a structure variable can be initialized however a structure must be declared as static if it is to be initialized inside a function.

Ex

```
main()
{
        static struct
        {
                int weight;
                float height;
        }student={60,180.65};
--------------
--------------
}
```

It assign the value 60 to student.weight and 180.65 to stud.height. there is one-to-one corresponding between the members and their initializing values.

A many type of variables is possible in initialization a structure. The following statements initialize two structure variables. In this it is essential to use tag name.

```
main()
{
        struct st_record
        {
                int weight;
                 float height;
        };
struct st_record student1={60,180.75};
struct st_record student2={53,170.60};
-------------
-------------
}
```

Another method to initialize a structure variable outside the function as shown follow.

```
struct stud_record
{
        int weight;
        float height;
}student1={60,180,75};
main()
{
        struct stud_record student 2={53,170.60};
        ------------------------
        ------------------------
}
```

C langue does not permits the initialization of individual structure members within the template. The initialization must be done only in the declaration of actual variables.

Array Of Structure:

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example.

struct class student[100];

define an array called student, that consist of 100 elements. Each element is defined to be of the type struct class. Consider the following

```
struct marks
{
        int subject1;
        int subject2;
        int subject3;
};
main()
{
        static struct marks student[3]={{33,44,55},{88,87,45},{57,87,90}}
}
```

This declares the student as an array of three element student[0], student[1],student[2] and initializes their members as follows.

```
student[0].subject1=33;
student[1].subject2=44;
--------------------------
--------------------------
```

Note that array is declared just as it would have been, with any other array. Since student is an array, we use the usual array accessing method to access individuals elements and then the member operator to access members.

An array of students is stored inside the memory in the same way as a multi-dimensional array. The array student actually looks like.

Student[0].subject1	33
.subject2	44
.subject3	55
Student[1].subject1	88
.subject2	87
.subject3	45
Student[2].subject1	57
.subject2	87
.subject3	90

The program is shown in following way. We have declared a four_member structure, the fourth one for keeping the student_totals. We have also declared an array total to keep the subject_totals and the grand_totals the grand_total is given by total.

Note that a member name can be any valid c name and can be the same as an existing structure variables name. the linked name total.toatl represent the total member of structure variables total.

```
//program of array of structure.
struct marks
{
        int sub1;
        int sub2;
        int sub3;
        int total;
};
main()
{
        int i;
        static struct marks student[3]={{40,55,45},
                                        {80,75,90},
                                        {75,65,100}};
        static struct marks total;
        for(i=0;i<=2;i++)\
        {
                student[i].total=student[i].sub1+student[i].sub2+student[i].sub3;
                total.sub1=total.sub1+student[i].sub1;
                total.sub2=total.sub2+student[i].sub2;
                total.sub3=total.sub3+student[i].sub3;
        }
        printf("Stuent          Total\n\n");
        for(i=0;i<=2;i++)
        {
                printf("\n Subject          Total\n\n");
                printf(%s    %d\n%s  %d\n",i+1,student[i].toatl);
                printf("subject 1=%s',total.sub1);
                printf("subject 2=%s',total.sub2);
                printf("subject 3=%s',total.sub3);
                printf("\n Grand total =%d \n',total.total);
        }
        getch();
}
```

Structure within Structure:

Structure within the structure means nesting of structure. Nesting of a structure is permitted in C.

The following structure defined to store the information about the salary of employees.

```
struct salary
{
        char name[20];
        char department[10];
        int basic_pay;
        int daily_allowance ;
        int house_allowance;
}employee;
```

This structure defines name, department, basic_pay, and two kinds of allowance. We can group all the items related to allowance together and declare them under a structure as shown below.

```
struct salary
{
        char name[20];
        char department[10];
        struct
        {
                int basic_pay;
                int daily_allowance;
                int hourse_allowance;
        }allowance;
}employee;
```

The salary structure contains a member named allowance which itself is a structure with three members. The members contained in the inner structure namely daily, house and basic can be referred to as

```
employee.allowance.daily_allowance
employee.allowance.house_allowance
employee.allowance.basic_pay
```

An inner most member in a nested structure can be accessed by changing all the concerned structure variable with the member using dot operator

The following are some invalid

```
employee.allowance   //member is missing
 employee.house      //inner struct variable missing
```

An inner structure can have more than one variable. The following is

```
struct salary
{
----------
```

```
----------
struct
 {
        int daily_allow;
}allowance,arrears;
}employee[100];
```

The inner structure has two variable, allowance and arrears. This implies both of them have the same structure template. Note the comma after the name allowance. A base member can be accessed as follows.

```
employee[1].allowance.daily
employee[1].arrears.daily
```

We can also use the tag name to define inner structure.

```
struct pay
{
        int daily;
        int hourse_rent;
        int city;
};
struct salary
{
        char name[20];
        char dept[10];
        struct pay allowance;
        struct pay arrears;
};
struct salary employee[100];
```

pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

It is also permissible to nest more than one type of structure.

```
Struct personal_record
{
        struct name_part name;
        struct addr_part address;
        struct date date_birth;
};
struct personal_record personal;
```

The first member of this structure is name which is of the type struct name_part. Similarly others members have their structure types.

```
#include<stdio.h>
#include<conio.h>
void main()
{
struct address
{
char add1[20];
char add2[20];
```

```
char add3[20];
char city[20];
long pin;
}a;
struct employee
{
char name[20];
int sal,daily,hra,pf,net_sal;
struct address a;
}emp;
clrscr();
printf("Enter the employee name and basic :");
scanf("%s%d",emp.name,&emp.sal);
emp.daily=emp.sal*.20;
emp.hra=emp.sal*.20;
emp.pf=emp.sal*.20;
emp.net_sal=emp.sal+emp.daily+emp.hra-emp.pf;
printf("\nEnter address 1");
scanf("%s",emp.a.add1);
printf("\nEnte address 2");
scanf("%s",emp.a.add2);
printf("\nEnter address 3");
scanf("%s",emp.a.add3);
printf("\nEnter the city");
scanf("%s",emp.a.city);
printf("\nEnter picode");
scanf("%ld",emp.a.pin);
printf("\n\nEmployee information");
printf("\nEmployee name      : %s",emp.name);
printf("\nBasic salary       : %d",emp.sal);
printf("\nda                 : %d",emp.daily);
printf("\nhra                : %d",emp.hra);
printf("\npf                 : %d",emp.pf);
printf("\nnet salary     : %d",emp.net_sal);
printf("\naddress1       : %s",emp.a.add1);
printf("\naddress2       : %s",emp.a.add2);
printf("\naddress3       : %s",emp.a.add3);
printf("\nDity               : %s",emp.a.city);
printf("\npin code           : %ld",emp.a.pin);
getch();
}
```

Pointer And Structure

When we create an object of the structure the compiler can allocate the memory space the object of the structure. Some time we need to store address of object of the structure the we should use

pointer and we can access member of the structure using pointer variable, pointer to operator and member of the structure.

```
Struct tagname
{
-----
-----
}object,*object=&object;

Void main()
{
Printf("Specifiers",pointerobject->member);
}
```

Ex

```
Struct test
{
Int x;
Int y;
}a,*p=&a;

Void main()
{
Int c;
Clrscr();
p->x=10;
p->y=20;
c=p->x+p->y;
printf("\nAdd=%d",c);
getch();
}
```

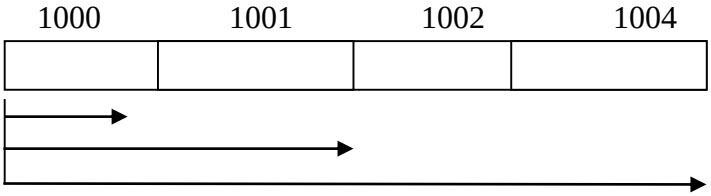
Introduction to Union:

Unions are a concept borrowed from structure and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows1.

```
union item
{
        int m;
        float x;
        char c;
}code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable.

Storage of 4 bytes



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members.

To access a union member, we can use the same syntax that we use for structure members.

```
code.m
code.x
code.c
```

are all valid membe variables. During accessing , we should make sure that we are accessing the member whose value is currently stored.

```
Code.m=455;
Code.x=456.66;
Printf("%d",code.m);
```

In effect, an union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value.

Ex:-

```
main()
{
        union index
        {
                int a;
                float b;
                char c;
        };
        union index e;
        e.a=30;
        printf("%d",e,a);
        e.b=44.55;
        printf("%d",e.b);
        e.c='A';
        printf("%c",e.c);
        getch();
}
```

Chapter 6
File Management in C

Introduction:

Until now we have been using the function such as scanf and printf to read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volume of data and in such situations, the console oriented I/O operations pose two major problem.

1. It become time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of file to store data. A file is a place on the disk where a group of related data is stored. Like most other language, C supports a number of functions that have ability to perform basic file operations.

Defining And Opening A File:

If we want to store data in a file in the secondary memory, we must specify certain things about the file, the operating system. They include.

File name
Data structure
Purpose

Filename is a string of character that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension.

Input.data
Store
Prog.c
Student.c
Text.out

Data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore all file should be declared as type FILE before they are used. FILE is a defined data type.

When we open a file, we must specify what we want to do with the file. Ex we may write data to the file or read the already existing data.

Syntax:

FILE *file pointer variable name;
File pointe variable name=fopen("Filename","mode");

The first statement declare the variable fp as a pointer to the data type FILE. FILE is the structure that is defined in the I/O library, the second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening this file. The mode does this job.

r open the file for reading only.
w open the file for writing only.

a open the file appending data to it.
r+ the existing file is opened to the beginning for both reading and writing.
w+ same as w except both for reading and writing.
a+ same as a except both for reading and writing.

Note that both file name and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen.

1. When the mode is 'writing' a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending' the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the file is opened with the current contents safe otherwise an error occurs.

Ex:

```
FILE *p1, *p2;
P1=fopen("data","r");
P2=fopen("result","w");
```

The file data is opened for reading and results is opened for writing. In case, the results file already exists, its contents are deleted and the file is opened as a new file. If data file does not exist, an error will occur.

Closing a File:

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out for m the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. So that's why we must closing unwanted files. Another instance where we have to close a file is when we want to reopen the same file in a different mode.

Syntax:

```
fclose(file_pointer);
```

ex:

```
--------------
FILE *p1, *p2;
P1=fopen("INPUT","w");
P2=fopen("OUTPUT","r");
---------------
fclose(p1);
fclose(p2);
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file. A matter of fact all files are closed automatically whenever a program terminates.

Input/ Output Operation On File:

Once a file is opened, reading out of or writing to it accomplished using the standard I/O

The getc and putc functions

The simplest file I/O functions are getc and putc. These are analogous to getchar and putchar functions and handle one character at a time. Assume that a file is opened with mode w and file pointer fp1.

```
putc(c,fp1);
```

writes the character contained in the character variable c to the file associated with FILE pointer fp1. Similarly getc is used to read a character from a file that has been opened in read mode.

```
c=getc(fp2);
```

would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of getc or putc. The getc will return an end-of-file marker EOF, when end of the file has been reached. Therefore the reading should be terminated when EOF is encountered.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *f;
    char c;
    printf("DATA input\n");
    f=fopen("INPUT","w");
    while((c=getchar())!=EOF)
    putc(c,f);
    fclose(f);
    printf("OUT put data\n\n");
    f=fopen("INPUT","r");
    while((c=getc(f))!=EOF)
    printf("%c",c);
    fclose(f);
    getch();
}
```

The getw and putw functions

The getw and putw are integer-oriented functions. They are similar to the getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of getw and putw are.

```
putw(integer, fp);
getw(fp);
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *f1,*f2,*f3;
```

```
int number,i;
printf("Enter the any number \n");
f1=fopen("data.txt","w");
for(i=0;i<=30;i++)
{
scanf("%d",&number);
if(number==-1) break;
putw(number,f1);
}
fclose(f1);
f1=fopen("data.txt","r");
f2=fopen("ODD.txt","w");
f3=fopen("EVEN.txt","w");
while((number=getw(f1))!=EOF)
{
if(number%2==0)
putw(number,f3);
else
putw(number,f2);
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("ODD.txt","r");
f3=fopen("EVEN.txt","r");
printf("\n\nContents of ODD file\n\n");
while((number=getw(f2))!=EOF)
printf("%4d",number);
printf("\n\ncontents of EVEN file\n\n");
while((number=getw(f3))!=EOF)
printf("%4d",number);
fclose(f2);
fclose(f3);
getch();
}
```

The fprintf and fscanf functions

We have seen functions, that can handle only one character or integer at a time. Most compiler support two other functions, namely fprintf and f scanf, that can handle a group of mixed data simultaneously.

The function fprintf and fscanf perform I/O operations that are identical to the familiar printf and scanf functions, except of course that they work on file. The first argument of these functions is a file pointer which specifies the file to used. The general form of fprintf is

fprintf(fp,"control string",list);

Where fp is a file pointer associated with a file that has been opened for writing. The control string contains output specifications for the items in the list. The list may include variables, constants and string.

fprintf(f1,"%s %d %f",name,age,7.5);

The general format of the fscanf function is

fscanf(fp,"Control string",list);

fscanf(f2,"%s %d ",item,&quantity);

```
void main()
{

    FILE *fp;
    int number,quentity,i;
    float price,value;
    char item[10],filename[10];
    printf("Input file name \n");
    scanf("%s",filename);
    fp=fopen("filename","w");
    printf("Input inventory data\n\n");
    printf("Item name   number   price   quentity\n");
    for(i=0;i<=3;i++)
    {
        fscanf(stdin,"%s %d %f %d",item,&number,&price,&quentity);
        fprintf(fp,"%s %d %.2f %d",item,number,price,quentity);
    }
    fclose(fp);
    fprintf(stdout,"\n\n");
    fp=fopen(filename,"r");
    printf("Item name   number   price   quentity\n");
    for(i=0;i<=3;i++)
    {
        fscanf(fp,"%s %d %f %d",item,&number,&price,&quentity);
        value=price*quentity;
        fprintf(stdout,"%s %d %f %d",item,number,price,quentity,value);
    }
    fclose(fp);
    getch();
}
```

Error handling during I/O operations:

It is possible that an error may occur during I/O operations on a file. Typical error situations include

1. Trying to read beyond end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.

5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result is termination of the program or incorrect output. Feof and ferror that can help us detect I/O errors in the files.

The feof function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If fp is a pointer to file that has just been opened for reading, then the statement

```
if(feof(fp));
printf("End of data. \n");
```

The ferror function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement.

```
if(ferror(fp)!=0)
printf("An error has occurred");
```

We know that whenever a file is opened using fopen function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not.

```
if(fp==NULL)
printf("File could not be opened.\n");
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char *filename;
        FILE *fp1,*fp2;
        int i,number;
        fp1=fopen("TEST","w");
        for(i=0;i<=100;i+=10)
        putw(i,fp1);
        fclose(fp1);
        printf("\nInput filename\n");
        open_file:
        scanf("%s",filename);
        if((fp2=fopen(filename,"r"))==NULL)
        {
                printf("Cannot open the file.\n");
                printf("The filename again\n\n");
                goto open_file;
        }
        else
                for(i=0;i<=20;i++)
                {
                        number=getw(fp2);
                        if(feof(fp2))
```

```
                        {
                                printf("\nRan out of data\n");
                                break;
                        }
                        else
                                printf("%d\n",number);
                }
        fclose(fp2);
        getch();
}
```

Random access to file:

So far we have discussed file function that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions fseek, ftell and rewind available in the I/O library.

ftel takes a file pointer and return a number of type long, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following

```
n=ftell(fp);
```

rewind takes a file pointer and resets the position to the start of the file.

Ex

```
rewind(fp);
n=ftell(fp);
```

would assign 0 to n because the file position has been set to the start of the file by rewind .

fseek:- function is used to move the file position to a desired location within the file. It takes the following form.

```
fseek(file_ptr, offset, position)
```

file_ptr is a pointer to the file concerned, offset is a number or variable of type long. And position is an integer number. 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 of file
1	current position
2	end of file

The offset may be positive, meaning move forwards, or negative, meaning move backwards.

```
void main()
{
        FILE *fp;
        long n;
        char c;
        fp=fopen("RANDOM","w");
        while((c=getchar())!=EOF)
        putc(c,fp);
        printf("No of character entered =%d\n",ftell(fp));
        fclose(fp);
```

```
        fp=fopen("RANDOM","r");
        n=0L;
        while(feof(fp)==0)
        {
                fseek(fp,n,0);
                printf("Position of %c is %d\n",getc(fp),ftell(fp));
                n=n+5L;
        }
        putchar('\n');
        fseek(fp,-1L,2);
        do
        {
                putchar(getc(fp));
        }while(!fseek(fp,-2L,1));
        fclose(fp);
        getch();
}
```