Unit-5

15:10 PM
Tue - 23-Jan-2024.

Functions & File Handling.

Introduction to Functions, Function Declaration and Definition, Function call, Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters, Scope and Lifetime of Variables, Basics of File Handling.

Introduction to Functions:-

C functions are easy to define and use. Functions have been primarily limited to the three functions namely main, printf, and scanf.

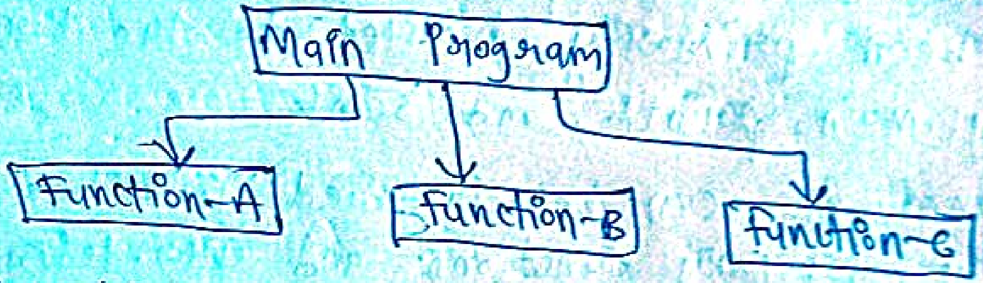
C functions can be classified into two categories, namely library functions and user-defined functions.

"main" is an example of user-defined function. printf and scanf belong to the category of library functions.

The main distinction b/w these two categories is that library functions are not required to be written by us, a user-defined function has to be developed by the user at the time of writing a program.

Every program must have a main function to indicate where the program has to begin its execution.

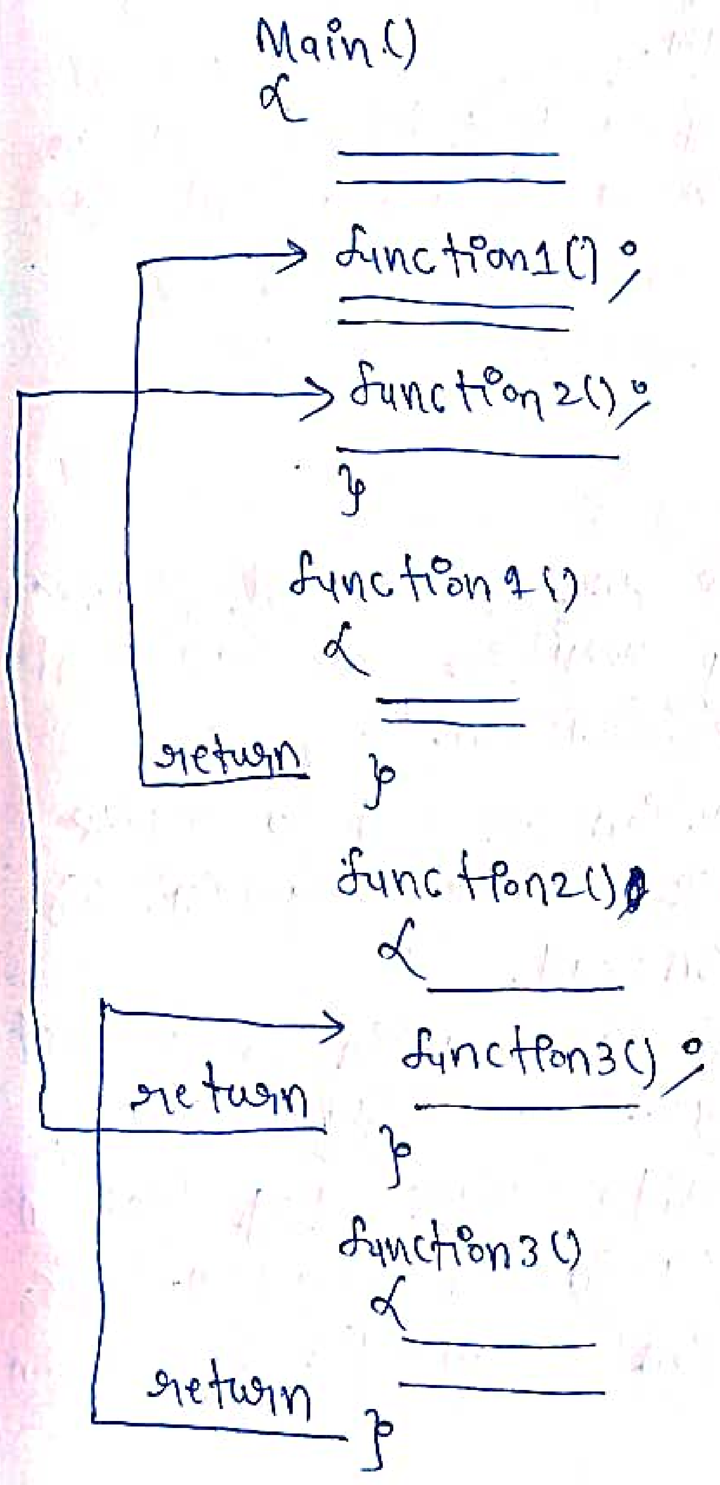
If a program is divided into functional parts then each part may be independently coded and later combined into a single unit.
These independently coded programs are called subprograms that are much easier to understand, debug, and test. In C, sub-programs are referred to as "functions".
To design a function that can be called and used whenever required. This saves both time and space.
The length of a program can be reduced by using functions at appropriate places.



Multi-function Program:

A function is a self-contained block of code that performs a particular task.
Once a function has been designed and packed, it can be treated as a "black box" that takes some data from the main program and returns a value.
Every C program can be designed using a collection of these black boxes known as functions.
The program execution always begins with the main function.

Any function can call any other function. It can call itself. A called function can also call another function. A function can be called more than once.



```
Main()
{
    ________
    ________
    function1();
    ________
    ________
    function2();
    ________
}
function1()
{
    ________
    ________
}
function2()
{ ________
    function3();
    ________
}
function3()
{ ________
    ________
}
```

Elements of User-defined functions:

Functions are classified as one of the derived data types in C.

→ Both function names and variable names are considered identifiers.

→ Like variables, functions have types associated with them.

→ Like variables, function names and their types must be declared and defined before they are used in a program.

In order to make use of a user-defined functions, we need to establish three elements that are related to functions.

(i) Function definition.
(ii) Function call
(iii) Function declaration.

The function definition is an independent program module that is specially written to implement the requirements of the function.

In order to use this function we need to invoke it at a required place in the program. This is known as the function call.

The program that calls the function is referred to as the calling program (or) calling function.

The calling program should declare any function that is to be used later in the program. This is known as the function declaration (or) function prototype.

Definition of Functions :-

A function definition, also known as function implementation.

→ Function name

→ Function type
→ list of parameters.
→ local variable declaration.
→ Function statements and
→ a return statement.
All the 6 elements are grouped in to two parts.
(i) Function header (first three elements)
(ii) Function body (second three elements).
A general format of function definition is

```
function_type function_name(parameter list) ←  function header
{
   local variable declarations;  ]
   executable statement;         ]
   - - - - -                     ]  function body
   - - - - -                     ]
   return statement;             ]
}
```

Function Header:

The function type specifies the type of value like float (or) double that the function is expected to return to the program calling the function.
If the function is not returning anything, then we need to specify the return type as void.
void is the fundamental data type in C.

Formal Parameter List:

The parameter lists declares the variables that will receive the data sent by the calling program.

These parameters can also be used to send values to the calling programs.

The parameters are also known as arguments.

The parameters list contains declaration of variables separated by commas and surrounded by parameters.

eg: int sum(int a, int b) {---}
float mul(float x, float y) {---}

Remember that there is no semicolon after the paranthesis.

The declaration of parameters can not be combined

eg: int sum(int a,b) X. (illegal).

Function Body:

The function body contains the declarations and statements. The body enclosed in braces.

when a function reaches its return statement the control is transferred back to the calling program.

In the absence of a return statement, the closing braces acts as a void statement.

A local variable is a variable that is defined inside a function..

Return Values and Their Types:-

The return statement can take one of the following forms:

return ; (or) return (expression);

When a return is encountered, the control is immediately passed back to the calling function.

```
eg: if (error)              eg: int p;
        return;                 p = x*y;
                                return (p);

eg: if (x<=0)
        return 0;
    else
        return 1;
```

All the functions by default return int type data.

Function Calls:

A function can be called by simply using the function name followed by a list of actual parameters (or) arguments.

When the compiler encounters a function call, the control is transferred to the function_name. This function is executed line by line.

```
eg: main ()
 {
   int y;                    Actual parameters.
   y = mul (10,5);        // function calling.
 }
int mul (int a, int b)    formal parameters.
 {
   int p;        // local variable.
   p = a*b;
   return p;
 }
```

Note:

→ If the actual arguments are more than the formal parameters, the extra actual arguments will be discarded.

→ If the actual s are less than the formals, the unmatched formal arguments will be initialized to some garbage.

Function declaration also known as function prototype.

→ The parameter list must be separated by commas

→ The parameter names do not need to be the same in the prototype declaration and the function definition.

→ The types must match the types of parameters in the function.

A prototype declaration may be placed in two places in a program:
(i) Above all the functions. (Including main)
(ii) Inside a function definition.

When we place the declaration above all the functions the prototype is referred to as a global prototype.

When we place a function definition the prototype is called a local prototype.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the scope of the function.

It is a good programming style to declare prototypes in the global declaration section before main.

Parameters:
Parameters also known as arguments are used in the following three places.
(i) In declaration (or) prototype.
(ii) In function call
(iii) In function definition.

The parameters used in the prototype and function definitions are called formal parameters and those used in function call are called actual parameters.

Category of Functions:

A function depending on whether arguments are present (or not) and whether a value is returned (or not), may belong to one of the following categories.

(i) Functions with no arguments and no return values

(ii) Functions with arguments and no return values

(iii) Functions with arguments and one return value

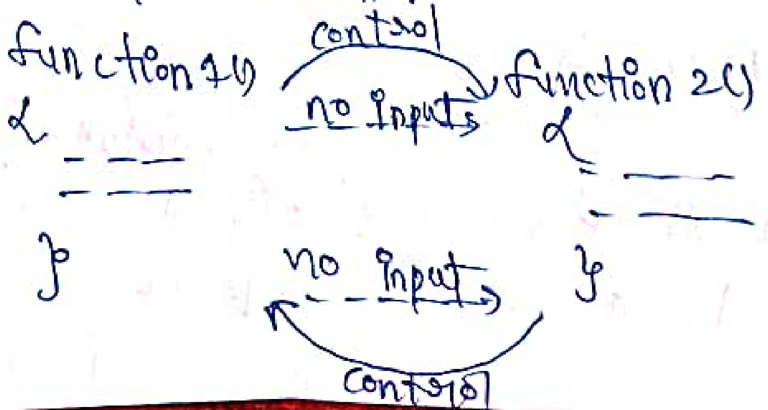
(iv) Functions with no arguments but return a value.

(v) Functions ~~[illegible]~~ that return multiple values.

No arguments and No return value:

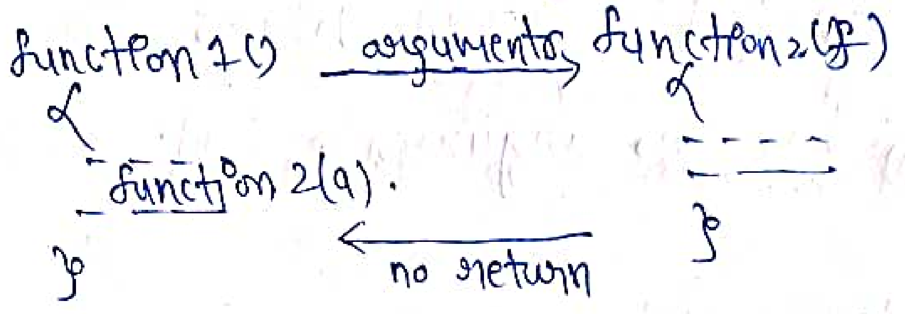
When a function has no arguments, it does not receive any data from the calling function. When it does not return a value, the calling function does not receive any data from the called function.

There is no data transfer b/w the calling function and the called function.

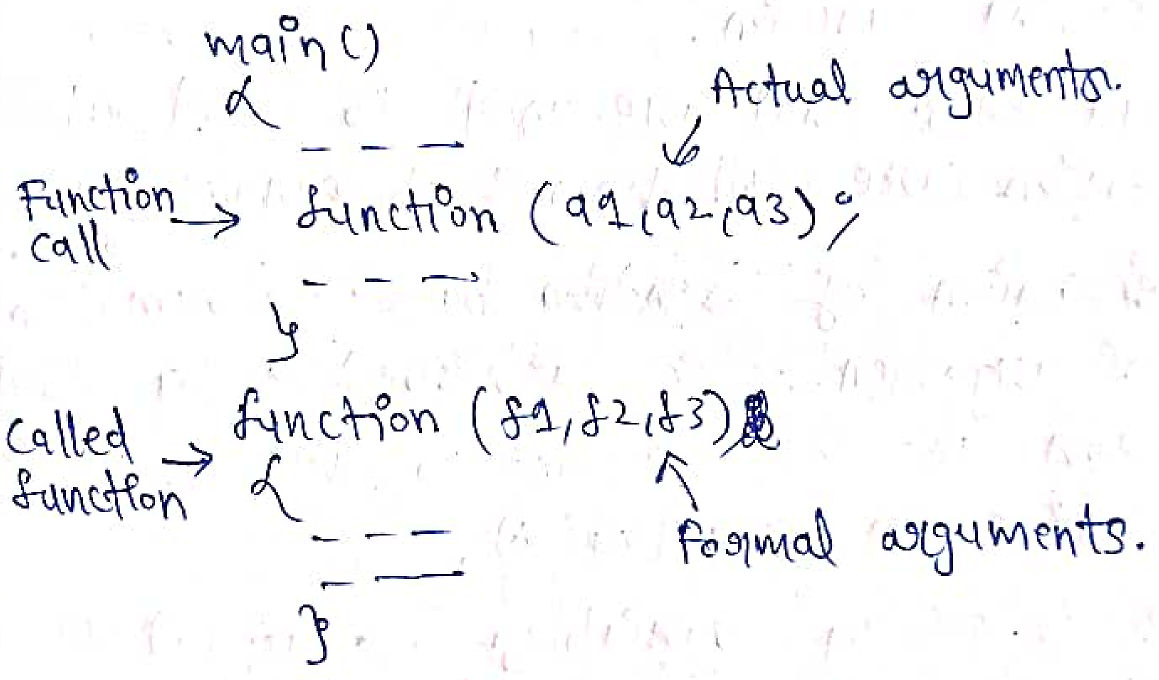


Arguments But No Return Values:

Function has arguments as its parameters, but it does not return any value.

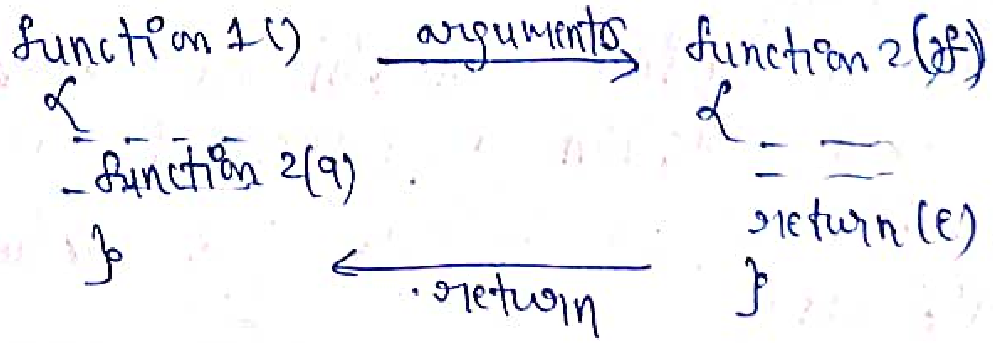


when a function call is made, only a copy of the values of actual arguments is passed into the called function.



Arguments with Return Value:

Function has arguments / parameters, ~~but~~ and it also has a return value.



C function returns a value of the type int as the default case when no other type is specified explicitly.

No arguments but a Return value:

The function has no arguments but returns a value.

Return Multiple Values:

In C, using the arguments not only to receive information but also to send back information to the calling function.

The arguments that are used to "send out" information are called output parameters.

The mechanism of sending back information through arguments is athieved using what are known as the address operator (&) and indirection operator (*).

The use of pointer variables as actual parameters for communicating data b/w functions is called "pass by pointers" (or) call by address (or) reference".

Recursion:

When a called function in turns calls another function a process of chaining occurs.

Recursion is a special case, where a function calls itself.

eg: factorial (int n)

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

The Scope, Visibility and Life time of Variables:-

In a BASIC program a variable retains, its value throughout the program. It is not always the case in C.

In C not only do all variables have a data type, they also have a storage class.

The following variable storage classes are most relevant to functions:

→ Automatic variables
→ External variables
→ Static Variables
→ Register variables.

The scope of variable determines over what region of the program a variable is actually available for use (active).

Longevity refers to the period during which a variable retains a given value during execution of a program (alive).

The visibility refers to the accessibility of a variable from the memory.

The variables may also be broadly categorized, depending on the place of their declaration, as internal (local) or external (global).

Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

Automatic Variables:

Automatic variables are declared inside a function in which they are to be utilized.

They are created when the function is called and destroyed automatically when the function is exited.

Automatic variables are private / local to the function in which they are declared.

Automatic variables are also referred to as local or internal variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable.

We may also use the keyword auto to declare automatic variables explicitly.

```
eg: main()                    eg: main()
    {                             {
      int number;                   auto int number;
      ----                          -----
      ----                          -----
    }                             }
```

One important feature of automatic variables is that their value can not be changed accidentally.

Any variable local to main will be normally alive throughout the whole program, although it is active only in main.

During recursion, the nested variables are unique auto variables.

External Variables:

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables.

Unlike local variables, global variables can be accessed by any function in the program.

External variables are declared outside a function.

```
eg: int number = 5;
    main()
    {----
      ----
    }
```

Once a variable has been declared as global, any function can use it and change its value.

External variables can be declared by using "extern" keyword.

eg: main()
```
{
    extern int y;
    ---
    ---
}
```

The distinction b/w definition and declaration also applies to functions.
A function is defined when its parameters and function body are specified. This tells the compiler to allocate space for the function code and provides type information for the parameters.
Functions are external by default.

Static Variables:

The value of the static variables persists until the end of the program. A static variable can be declared by using the keyword "static".

eg: static int x;

A static variable may be either an internal type (or) an external type depending on the place of declaration.
Internal static variables used to retain values between function calls.

A static variable is initialized only once, when the program is compiled.
The difference b/w a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined, while the simple external variable can be accessed by other files.

__Register Variables:__

A register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs.

eg: register int count;

C will automatically convert register variables in to non-register variables once the limit is reached.
"register" keyword is used to declare the register variables.

__Nested Block:__

A set of statements enclosed in a set of braces is known a block or compound statement.
A block can have its own declarations and other statements.

It is also possible to have a block of such statements inside the body of a function or another block. is known as nested block.

```
eg: main()
    {                                 <-----
        int a=10;                          |
        int b=20;                          |
        {               <----              |   Outer
            int a=0;        |              |   block
            int c=a+b;      |  inner       |
            ----.           |  block       |
                        <----              |
        }                                  |
        b=a;                               |
    }                                 <-----
```

Modifying parameters inside functions using pointers:-

We can modify parameters inside functions by passing their address using pointers. This allows you to change the values of variables outside the function, effectively updating them.

```
eg: #include<stdio.h>
      void modifyValues(int *a, int *b) {
           *a = *a *2;
           *b = *b+5;
      int main() {
           int x=10;
```

```
    int y=7;
    print("Before function call: x=%d, y=%d\n",x,y);
    modifyValues (&x,&y);
    printf("After function call: x=%d, y=%d\n",x,y);
    return 0;
  }.
```

output:

```
  Befor function call : x=10, y=7.
  After function call : x=20, y=12.
```

Keep in mind that when passing parameters by pointers, you need to ensure that the pointers are valid (not NULL).

Arras as Parameters:-

In C, we can pass arrays as parameter to functions.

When you pass an array to a function, you are essential passing a reference to the array, allowing the function to work with the original array.

eg:
```
#include<stdio.h>
    void modifyArray (int arr[], int size) {
        for (int i=0; i<size; i++) {
            arr[i] *=2;
        }
    }
```

```
void printArray (int arr[], int size) {
        printf (" Array elements :");
        for (int i=0; i<size; i++) {
                printf(" %d", arr[i]);
        }
        printf("\n");
    }
int main() {
    int myArray [] = {1,2,3,4,5};
    int arraySize = sizeof (myArray) /
                             sizeof (myArray [0]);
    printf (" Before function call:\n");
    printArray(myArray, arraySize);
    modifyArray (myArray, arraySize);
    printf (" After function call:\n");
    printArray (myArray, arraySize);
    return 0;
}
```

Output:

```
Before function call:
Array elements : 1 2 3 4 5
After function call:
Array elements : 2 4 6 8 10
```

Note that when you pass an array to a function, you don't need to use the "&" (address-of) operator. The array name itself acts as a pointer to the first element of the array.

22:18PM
Tue - 23-Jan-2024 //.

Basics of File Handling :-

We have been using the functions 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).

The console oriented I/O operations pose two major problems:

(i) It becomes cumbersome and time consuming to handle large volumes of data through terminals.

(ii) The entire data is lost when either the program is terminated or the computer is turned off.

A file is a place on the disk where a group of related data is stored.

C supports a number of functions that have the ability to perform basic file operations, which include:

→ naming a file
→ opening a file
→ reading data from a file
→ writing data from a file
→ closing a file

there are 2 distinct ways to perform file operations in C. the first one is known as the low-level I/O and uses UNIX system calls. The second method is referred to as the high-level I/O operation and uses functions in C standard I/O library.

The most important file handling functions that are available in the C library are.,

fopen()	=	Creates a new file for use.
	=	opens an existing file for use
fclose()	=	Closes a file which has been opened for use
getc()	=	Reads a character from a file
putc()	=	writes a character to a file
fprintf()	=	writes a set of data values to a file
fscanf()	=	Reads a set of data values from a file.
getw()	=	Reads an integer from a file
putw()	=	writes an integer to a file
fseek()	=	sets the position to a desired point in the file
ftell()	=	Gives the current position in the file
rewind()	=	sets the position to the beginning of the file.

Defining and Opening a file:

To store data in a file in the secondary memory, we must specify certain things about the file to the operating system.

They include,

→ File name

→ Data Structure

→ Purpose

Filename is a string of characters that make up a valid filename for the OS.

It may contain two parts, a primary name and an optional period with the extension.

eg: input.data.

```
    program.c
    tent.out
```

Data structure of a file is defined as "FILE" in the library of standard I/o function definitions.

"FILE" is a defined data type.

The general format for declaring and opening a file:

```
FILE    *fp;
fp = fopen ("filename", "mode");
```

The 'fp' pointer, which contains all the information about the file is subsequently used as a communication link b/w the system and program.

The mode specifies the purpose of opening a file,

r	—	open the file for reading only
w	—	open the file for writing only.
a	—	open the file for appending (or adding data to it.

eg: FILE *p1, *p2 ;

```
p1 = fopen ("~~data~~ file1", "r");
p2 = fopen ("file2 ", "w");
```

If the file2 already exists, its contents are deleted and the file is opened as a new file. If file1 does not exist, an error will occur.

Additional modes of operations are,

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.

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 from the

buffers and all links to the file are broken.
It also prevents any accidental misuse of the file.

fclose (file_pointer);

Once a file is closed, its file pointer can be reused for another file.

<u>Input/output Operations on Files:</u>

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

<u>The getc and putc functions:</u>

· 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.

putc, writes the character contained in the character variable to the file associated with FILE pointer.

putc(c, fp1);

Ily, getc is used to read a character from a file that has been opened in read mode.

c = getc(fp2);

The file pointer moves by one character position for every operation of getc ~~and~~ (or) putc.
The getc will return an EOF - End of-File marker, when end of the file has been reached.

The reading should be terminated when EOF is encountered.

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.

The general forms of getw and putw are as follows:

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

The fprintf and fscanf functions:

Most compilers support two other functions, namely fprintf and fscanf that can handle a group of mixed data simultaneously.

The functions fprintf and fscanf perform I/O operations that are identical to the printf and scanf functions, except they work on files.

The general form of fprintf is

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

The general form of fscanf is

```
fscanf(fp, "control string", list);
```

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,

→ Trying to read beyond the EOF - End of File mark.
→ Device overflow
→ Trying to use a file that has not been opened.
→ Trying to perform an operation on a file, when the file is opened for another type of operation.
→ Opening a file with an invalid filename.

An unchecked error may result in a premature termination of the program or incorrect output.

We have two status - inquiry library functions : feof and ferror than can help us detect IO errors in the files.

The feof function can be used to test an end of the file condition.

It takes a file pointer as its argument and returns a non-zero integer value if all of the data from the specified file has been read, and returns zero otherwise.

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

The ferror function reports the status of the file indicated. It is same as feof.

eg: if (ferror(fp) != 0)
printf ("An error has occured \n");

Random Access to files:

"ftell" takes a file pointer and return a number of type long, that corresponds to the current position.

This function is useful saving the current position of a file.

eg: n = ftell (fp);

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

eg: rewind (fp);
n = ftell (fp);

"fseek" function is used to move the file position to a desired location within the file.

fseek (file_ptr, offset, position);

Command Line Arguments:

It is a parameter supplied to a program when the program is invoked.

In fact, main can take two arguments called argc and argv..

The variable argc is an argument counter that counts the number of arguments on the command line.

The argv is an argument vector and represents an array of character pointers that point to the command line arguments.
The size of the array will be equal to the value of the argc.

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

```
    {
      - - - -
      - - -
    }
```

09:44 AM
wed - 24 - Jan - 2024
[illegible]
