Revision by Topic

Introduction to Digital computers –

Super-Computer: These are used for super complex tasks. Such as weather forecasting, nuclear simulation etc. It was first introduced in 1976 and the name was CRAY 1.

Main Frame Computer: These are used by large corporations or organizations for large data processing, critical applications, industry and consumer statistics, enterprise resource planning and transactions processing.

Micro Computer: Small and relatively inexpensive computer with a microprocessor as its central processing unit (CPU) are called micro-computer. It includes a microprocessor, memory, and minimal input/output (I/O) circuitry mounted on a single printed circuit board.

Mini-Computer: A term no longer much used is Mini-Computer. It is a size of between micro and minicomputer.

Programming Language

Programming language is a formal language comprising a set of strings that produce various kinds of machine code output. Programming language is kind of a **Computer Language.** They are used in computer programming to implement algorithms.

Most programming languages consist of instructions for computers. There are programmable machine that use a set of specific instructions, rather than general programming languages. Since early 1800s, programs are being used for directing the behavior of machines such as Jacquard Looms, Music Boxes, and player pianos.

Flow Chart and Algorithm:

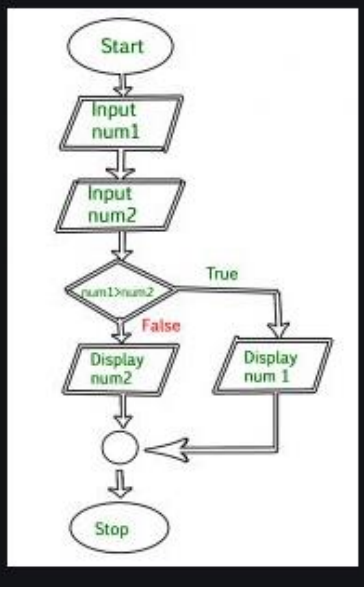
Algorithm: Algorithm is a process or set of rules to be followed in calculations or other problem-solving operations. Here is an example of Algorithm of Linear Search Program.

Algorithm of linear search :

```
1. Start from the leftmost element of arr[] and
one by one compare x with each element of arr[].
2. If x matches with an element, return the index.
3. If x doesn't match with any of elements, return -1.
```

Flowchart: A flowchart is a is a graphical representation of algorithm. Programmers use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of the information and processing.

Example: Draw a flowchart to input two numbers from the user and display the largest of two numbers



Here is the difference between Algorithm and Flowchart:

SL No.	Algorithm	Flowchart
1	Algorithm is step by step procedure to solve the problem	Flowchart is a diagram created by different shapes to show the flow of Data.
2	Algorithm is complex to understand	Flowchart is easy to understand.
3	In algorithm only text is used.	In flowchart, symbol/shapes are used.
4	Algorithm is easy to debug.	Flowchart is hard to debug.
5	Algorithm is difficult to construct.	Flowchart is simple to construct.
6	Algorithm does not follow any rules.	Flowchart follows rules to be constructed.
7	Algorithm is pseudo code for the program.	Flowchart is just graphical representation of that logic.

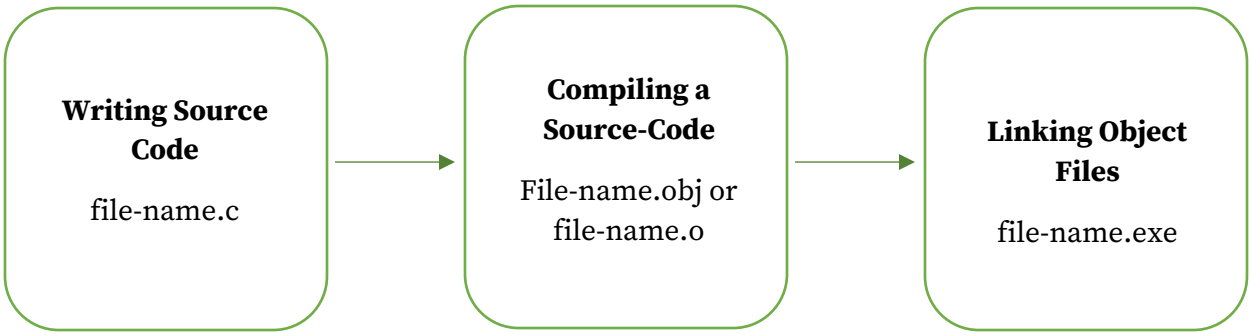
Structured Programming using C:

'C' is a structured programming language in which program is divided into various modules. Each module can be written separately and together it forms a single 'C' Program. This structure makes it easy for testing, maintaining and debugging processes.

Here are some Basic C Commands:

C Basic Commands	Explanation
#include <stdio.h>	This program includes standard input output header (stdio.h) from the C library before compiling a C program.
int main()	It is the main function from where C program execution begins.
{	Indicates the beginning of the main function.
/*_Some_Comments_*/	Whatever written inside the command "/**/" inside a C program. It will not be considered for compilation and execution.
printf("Hello World");	This command prints the output on the screen.
scanf("");	This command takes input from the user.
getch();	This command is used for any character input from the keyboard.
return 0;	This command is used to terminate the C program (main function) and it returns 0.
}	It is used to indicate the end of the main function.

The working process of C Programming Language:



Variables and Constants

Constants: The name constants are given to such variables or values in C programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any types of constants like integer, float, octal, hexadecimal, character constants etc. Every constant has some range. The integers that are too big to fit into an int will be taken as a long. Now there are various ranges that differ from unsigned to signed bits. Under the signed bit, the range of an int varies from -128 to +127 and under the unsigned bit, int varies from 0 to 255.

```
#include <stdio.h>
//constants
#define val 10;
#define floatval 4.5;
#define charval G;
//Driver Code
int main()
{
        printf("Integer constant: %d\n", val)
        printf("Floating point constant: %f\n", floatval);
        printf("Character Constant: %c\", charval);
        return 0;
}
```

Output:

Integer Constant: 10

Floating point constant: 4.500000

Character Constant: G

Variable: In simple terms variable is a storage place which has some memory allocated to it. Basically, a variable is used to store some form of data. Different types of variables require different amounts memory and have some specific set of operations which can be applied to them.

Variable Declaration:

A typical variable declaration is of the form:

type variable_name
or for multiple variables:
type variable1_name, variable2_name, variable3_name;

A variable name can consist of alphabets (both upper and lower case) numbers and the '_' underscore character. However, the name must not start with a number.

```
#include <stdio.h>
int main()
{
   // declaration and definition of variable 'a123'
   char a123 = 'a';

   // This is also both declaration
   // and definition as 'b' is allocated
   // memory and assigned some garbage value.
   float b;

   // multiple declarations and definitions
   int _c, _d45, e;

   // Let us print a variable
   printf("%c \n", a123);

   return 0;
}
```

Difference Between Variables and Constant:

SL No.	Constants	Variables
1	A value that cannot be altered throughout the program.	A storage location paired with an associated symbolic name which has a value.
2	It is similar to variable, but it cannot be modified by the program once defined	A storage area holds data.
3	Cannot be changed	Can be changed according to the need of the programmer.
4	Value is fixed	Value is varying.

Operators:

An operator is a symbol that operates on a value or a variable. For example: "+" is an operator perform addition.

C has a wide range of operators to perform various operations.

Arithmetic Operators:

SL No.	Operator	Meaning of Operator
1	+	Addition or unary plus.
2	-	Subtraction or unary minus.
3	*	Multiplication
4	/	Division
5	%	Remainder after division (modulo division)

Example of Arithmetic Operator

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
   int a = 9,b = 4, c;

   c = a+b;
   printf("a+b = %d \n",c);
   c = a-b;
   printf("a-b = %d \n",c);
   c = a*b;
   printf("a*b = %d \n",c);
   c = a/b;
   printf("a/b = %d \n",c);
   c = a%b;
   printf("Remainder when a divided by b = %d \n",c);

   return 0;
}
```

Output:
a + b = 13
a - b = 5
a * b = 36
3 / b = 2
Remainder when a divided by b = 1.

The operators '+', '-', '*', gives addition, subtraction and multiplication values as expected.

But in normal calculation a/b = 9/4 = 2.25. As the output variable 'c' is integer type so it ignores the '.25'. Hence, the output is 2.

The '%' modulo operator computes the remainder. When 'a=9' is divided by 'b=4'. The remainder is 1. The '%' modulo operator can only be used with integers.

C increment and decrement Operator

C has two operators increment '++' and decrement '- -' to change the value of an operand (Constant or variable) by 1.

Increment '++' operator increases the value by 1 whereas decrement '- -'operator decrease the value by 1. These two operators are unary operators, meaning they operate on a single operand.

Example

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
   int a = 10, b = 100;
   float c = 10.5, d = 100.5;

   printf("++a = %d \n", ++a);
   printf("--b = %d \n", --b);
   printf("++c = %f \n", ++c);
   printf("--d = %f \n", --d);

   return 0;
}
```

Output:
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000

Here, the operators used as prefixes. These operators also can be used as postfixes.

Assignment Operators:

An assignment operator is used for assigning a value to a variable. The most common assignment operator is '='

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a + b
-=	a -=b	a = a – b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

Example

```
// Working of assignment operators
#include <stdio.h>
int main()
{
  int a = 5, c;

  c = a;     // c is 5
  printf("c = %d\n", c);
  c += a;    // c is 10
  printf("c = %d\n", c);
  c -= a;    // c is 5
  printf("c = %d\n", c);
  c *= a;    // c is 25
  printf("c = %d\n", c);
  c /= a;    // c is 5
  printf("c = %d\n", c);
  c %= a;    // c = 0
  printf("c = %d\n", c);

  return 0;
}
```

Output:

```
c = 5
c = 10
c = 25
c = 5
c = 0
```

Relational Operator

A relational operator checks the relationship between two operands. If the relations is true it returns 1. If the relation is false it returns value 0.

Relational Operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is elevated to 0
>	Greater than	5 > 3 is elevated to 1
<	Less than	5 < 3 is elevated to 0
!=	Not Equal to	5!=3 is elevated to 1
>=	Greater than or equal to	5>=3 is elevated to 1
<=	Less than or equal to	5<=3 is elevated to 0

Example

```
// Working of relational operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10;

   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
   printf("%d < %d is %d \n", a, b, a < b);
   printf("%d < %d is %d \n", a, c, a < c);
   printf("%d != %d is %d \n", a, b, a != b);
   printf("%d != %d is %d \n", a, c, a != c);
   printf("%d >= %d is %d \n", a, b, a >= b);
   printf("%d >= %d is %d \n", a, c, a >= c);
   printf("%d <= %d is %d \n", a, b, a <= b);
   printf("%d <= %d is %d \n", a, c, a <= c);

   return 0;
}
```

Output:

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true.	If c = 5 and d = 2 then expression ((c == 5) && (d > 5)) equals to 0.
||	Logical OR. True only if either one operand is true.	If c = 5 and d = 2 then expression ((c ==5) || (d > 5)) equals to 1.
!	Logical NOT. True only if the operand is 0.	If c = 5, then expression !(c==5) equals to 0

Example:

```
// Working of logical operators

#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
   printf("(a == b) && (c > b) is %d \n", result);

   result = (a == b) && (c < b);
   printf("(a == b) && (c < b) is %d \n", result);

   result = (a == b) || (c < b);
   printf("(a == b) || (c < b) is %d \n", result);

   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) is %d \n", result);

   result = !(a != b);
   printf("!(a != b) is %d \n", result);

   result = !(a == b);
   printf("!(a == b) is %d \n", result);

   return 0;
}
```

Output:

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of Logical Operator Program:

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true.
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true)
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 false.
- !(a != b) evaluates to 1 because operand (a != b) is 0 false. Hence, !(a != b) is 1 true.
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

Bitwise Operator

During computation, mathematical operations like: addition, subtraction, multiplication, division etc. are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of Operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise complement
<<	Shift Left
>>	Shift Right

Other Operators:

Comma Operator:

Comma Operators are used to link related expressions together. For example

int a, c = 5, d;

The Sizeof Operator:

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc.).

Example

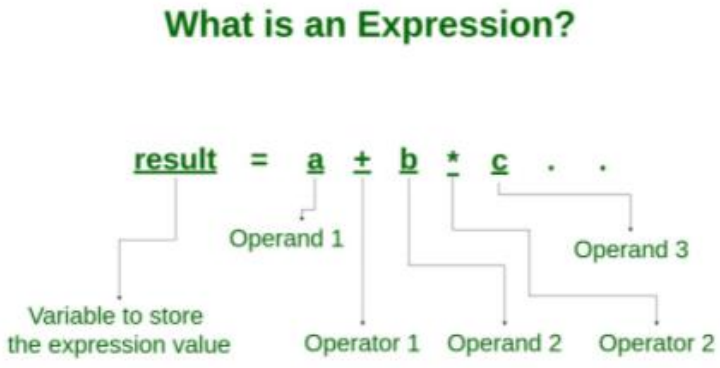
```
#include <stdio.h>
int main()
{
   int a;
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));

   return 0;
}
```

Output
Size of int = 4 bytes
Size of float = 4 bytes
Size of Double = 8 bytes
Size of Char = 1 byte

Expressions:

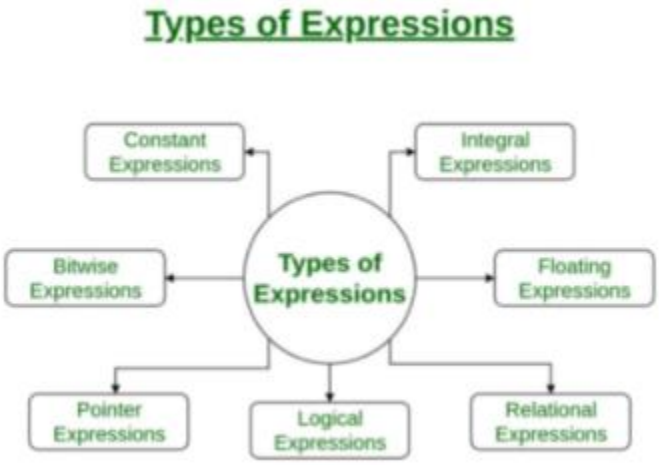
An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.



Example:
a+b
c
s-1/7*f
.
.
etc

Types of Expressions:

The following types are expressions:



Constant Expressions: Constant Expressions consists of only constant values. A Constant value is one that doesn't change.

Example:

- **5, 10+5 / 6.0, 'x'**

Integral Expressions: Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

Example:

- `x, x * y, x + int( 5.0)`

Floating Expressions: Float expressions are which produce floating point results after implementing all the explicit type conversions.

Example:

```
x + y, 10.75
```

Relational Expressions: Relational expressions yields result of type bool which takes a value true or false. When arithmetic operations are used on either side of a relational operator. They will be evaluated first and then the results compared. Relation expressions are also known as Boolean expressions.

Example:

```
x <= y, x + y > 2
```

Logical Expressions: Logical expressions combine two or more relational expressions and produces bool type results.

Example:

```
x > y && x == 10, x == 10 || y == 5
```

Pointer Expressions: Pointer expressions produce address values.

```
&x, ptr, ptr++
```

Bitwise Expressions: Bitwise expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.

Examples:

```
x << 3
```

Shifts 3 bits position to left.

```
y >> 1
```

Shifts 1 bit position to right.

Control Statements:

Control statements enable us to specify the flow of program control. The order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

There are four types of control statements in C:

- Decision making statements
- Selection Statements
- Iteration Statements
- Jump Statements

Decision making statement: if – else statement

The if – else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (whether the outcome is true or false)

Syntax

```
if (condition)
{
  statements
}
  else
{
  statements
}
```

Example of positive and negative number

```
#include<stdio.h>
int main()
   {
    int a;
    printf("n Enter a number:");
    scanf("%d", &a);
    if(a>0)
     {
      printf( "n The number %d is positive.",a);
     }
    else
     {
      printf("n The number %d is negative.",a);
     }
  return 0;
   }
```

Example of checking 2 Strings

```
#include <stdio.h>
#include <string.h>
int main()
   {
    char a[20] ,  b[20];
    printf("n Enter the first string:");
    scanf("%s",a);
    printf("n Enter the second string:");
    scanf("%s",b);
    if((strcmp(a,b)==0))
      {
       printf("nStrings are the same");
       }
    else
      {
      printf("nStrings are different");
      }
  return 0;
  }
```

The above program compares two strings to check whether they are same or not. The strcmp function is used for this purpose. It is declared in the string.h header file.

```
int strcmp(const char *s1, const char *s2);
```

Nested if and if – else statements

It is also possible to embed or to nest if-else statements one within the other. Nesting is useful in situations where one of several different courses of action to be selected.

The general format of a nested if – else statement is:

```
if(condition1)
{
// statement(s);
}
else if(condition2)
{
//statement(s);
}
.
.
.
.
else if (conditionN)
{
//statemenz(s);
}
else
{
//statement(s);
}
```

The above is also called if – else ladder. During the execution of nested if – else statement, as soon as a condition is encountered which evaluates to true, the statements associated with the particular if-block will be executed and the remainder of if-else statements will be bypassed. If neither of the conditions are true. Either the last else-block is executed or if-else block is absent, the control gets transferred to the next instruction present immediately after the else – if ladder.

The following program makes use of the nested if-else statement to find the greatest of three numbers.

Example:

```
#include<stdio.h>

int main( )
   {
       int a, b,c;
       a=6,b= 5, c=10;
       if(a>b)
          {
             if(b>c)
               {
                 printf("\nGreatest is: " , a);
               }
              else if(c>a)
                     {
                      printf("\nGreatest is: ", c);
                     }
            }
         else if(b>c)     //outermost if-else block
                {
                 printf("\nGreatest is: " , b);
                }
                else
                 {
                 printf( "\nGreatest is: " , c);
                 }
      return 0;
    }
```

The above program compares three integer quantities and prints the greatest. The first if statement compares the values of **a** and **b.** If **a>b** is true, program control gets transferred to the if-else statement nested inside the if block, where **b** is compared to **c.** If **b>c** is also true, the value of **a** is printed; else the value of **c** and **a** are compared and if **c>a** is true, the value of **c** is printed. After the rest of the if-else ladder is bypassed.

However, if the first condition **a>b** is false, the control directly gets transferred to the outermost else-if block, where the value of **b** is compared with the **c** (as **a** is not the greatest). If **b>c** is true, the value of **b** is printed else the value of **c** is printed. Note, the nesting, the use of braces, and the indentation. All this is required to maintain clarity.

Selection Statement: The switch case statement

A switch statement is used for **multiple way selections** that will be branch into different code segments based on the value of a variable or expression. This expression or variable must be of integer data type.

Syntax

```
switch (expression)
{
  case value1:
    code segment1;
    break;
  case value2:
    code segment2;
    break;
.
.
.
  case valueN:
    code segmentN;
    break;
  default:
    default code segment;
}
```

The value of this **expression** is either generated during program execution or read in as a user input. The case whose value is the same as that of the **expression** is selected and executed. The optional **default** label is used to specify the code segment to be executed when the value of the expression does not match with any of the case values.

The **break** statement is present at the end of every case. If it were not so, the execution would continue on into the code segment of the next case without even checking the case value. For example, supposing a **switch statement** has five cases and the value of the third case matches the value of expression. If no **break statement** were present at the end of the third case, all the cases after case 3 would also get executed along with case 3. If break is present only the required case is selected and executed. After which the control gets transferred to the next statement immediately after the switch statement. There is no **break** after **default** because the default case the control will either way get transferred to the next statement immediately after switch.

Example to print the day of the week:

```
#include<stdio.h>

int main( )

    {

     int day;

     printf("nEnter the number of the day:");

     scanf("%d",&day);

     switch(day)
        {
          case 1:
                  printf("Sunday");
                  break;
          case 2:
                  printf("Monday");
                  break;
          case 3:
                  printf("Tuesday");
                  break;
          case 4:
                  printf("Wednesday");
                  break;
          case 5:
                  printf("Thursday");
                  break;
          case 6:
                  printf("Friday");
                  break;
          case 7:
                  printf("Saturday");
                  break;
          default:
                  printf("Invalid choice");
        }
   return 0;

     }
```

This is a very basic program to explain the working procedure of the switch – case construct. Depending upon the number entered by the user, the appropriate case is selected and executed. For example, if the user input is 5, the case 5 will be executed. The **break** statement present in case 5 will pause execution of the switch statement

after case 5 and the control will get transferred to the next statement after switch which is:

return 0;

If you need to select among a large group of values, a switch statement will run much faster than a set of nested ifs. The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression.

The switch statement must be used when one needs to make a choice from a given set of choices. The switch case statement is generally used in menu-based applications. The most common use of a switch-case statement is in data handling or file processing.

Example: A switch case statement used in data file processing

```
#include<stdio.h>
int main()
    { //create file &set file pointer .
     int choice;
     printf("n Please select from the following options:");
     printf("n 1. Add a record at the end of the file.");
     printf("n 2. Add a record at the beginning of the file:");
     printf("n 3. Add a record after a particularrecord:";
     printf("nPlease enter your choice:(1/2/3)?");
     scanf("%d",&choice);
     switch(choice)
         {
           case 1:
                   //code to add record at the end of the file
                   break;
           case 2:
                   //code to add record at the beginning of the file
                    break;
           case 3:
                   //code to add record after a particular record
                    break;
           default:
                   printf("n Wrong Choice");
        }
    return 0;
    }
```

The above example of switch case generally involves nesting the switch-case construct inside an iteration construct like do-while.

Functions:

A function is a group of statements that together perform a task. Every C program has at least one function which is **main()** function and all the most trivial programs can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining A Function:

The general form of a function definition in C programming language is as follows: -

```
return_type function_name (parameter list) {
      body of the function
}
```

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function.

- **Return Type –** A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is keyword void.
- **Function Name –** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters –** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function body –** The function body contains a collection of statements that define what the function does.

Example –

```
/* function returning the max between two numbers */
int max (int num1, int num2) {
      /*local variable declaration */
      int result;
      if(num1 > num2)
            result = num1;
      else
            result = num2;
      return result;
}
```

Function Declarations:

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts –

```
      return_type function_name (parameter list);
```

For the above defined function max(), the function declaration as follows:

```
      int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required. So, the following is also a valid declaration:

```
      int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the topo of the file calling the function.

Calling a Function:

While creating a C function, you gave a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function you need to pass the required parameters along with the function name and if the function returns a value you can store the returned value.

For example –

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

We have kept **max()** long with **main()** and compile the source code. While running the final executable it produce the following result.

```
Max Value is: 200
```

Function arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function.

Sl No.	Call Type & Description
1.	**Call by value:** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2.	**Call by Reference:** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

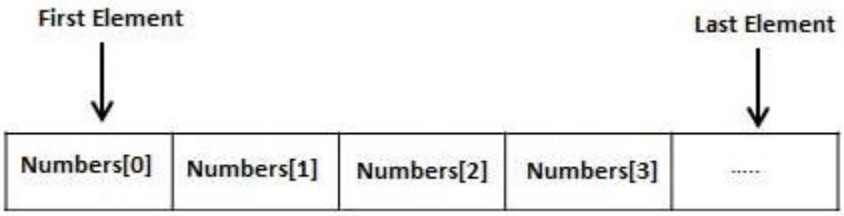
By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Arrays

Arrays a kind of data structure that can store a fixed size sequential collection of elements of the same type. An array is used to store a collection of data, but it is more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ... and number99. You can declare one array variable such as **numbers** and use **numbers[0], numbers[1]** and .... **Numbers[99]** to represent individual variables. A specific element of an array is accessed by an index.

All arrays consist of contiguous memory location. The lowest address corresponds to the first element and the highest address the last element.



Declaring Arrays:

To declare an array in C, a programmer specifies the type of elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare 10 – element array called **balance** of type Double, use this statement –

```
double balance[10];
```

Here **balance** is a variable array which is sufficient to hold up to 10 double numbers.

Initializing arrays

You can initialize an array in C either one by one or using single statement as follows –

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces {} cannot be larger than the number of elements that we declare for the array between square brackets [].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write -

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array.

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

Accessing Array Elements:

An element is accessed by indexing an array name. This is done by placing the index of the element within square brackets after the name of the array.

For Example:

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign he value to salary variable. The following example shows how to use all the three above mentioned concepts viz. declaration, assignment and accessing arrays –

```c
#include <stdio.h>

int main () {

   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Arrays in Detail

The following important concepts related to array should be clear to a C programmer –

Sl No.	Concept & Description
1	Multi-dimensional arrays: C supports multi-dimensional arrays. The simplest form of the multidimensional array is two-dimensional array.
2	Passing Arrays to functions:

	You can pass the function pointer to an array by specifying the array's name without an index.
3	Return Array from a function: C allows a function to return array.
4	Pointer to an array: You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Pointers:

Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers.

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

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

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are pointers?

A pointer is a variable whose value is the address of another variable. Such as direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var_name** is the name of the pointer variable. The asterisk used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise is the same a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use pointers?

a) We define a pointer variable.
b) Assign the address of a variable to a pointer and
c) Finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Example:

```c
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Null Pointers:

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

The NULL pointer is a constant with a value of zero defined in several standard libraries.

Consider the following program.

```c
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

When the above code is compiled and executed:

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

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

To check for a null pointer

```c
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Pointers in detail:

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer.

Sl No.	Concept & Description
1	Pointer Arithmetic: There are four arithmetic operators that can be used in pointers: ++, --, +, -
2	Array of Pointers: You can define arrays to hold a number of pointers.
3	Pointer to pointer: C allows you to have pointer on a pointer and so on.
4	Passing pointers to functions in C: Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5	Return pointer from functions in C: C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

Structure Unions:

Structure is user defined data type available in C that allows to combine data items of different kinds.

Structure are used to represent a record. Suppose you want to keep track your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure you must use the struct statement. The struct defines a new data type, with more than one member. The format of the struct statement is as follows: -

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as **int i**; or **float f;** any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book Structure -

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type.

Example

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function arguments:

You can pass a structure as a function argument in the same way you pass any other variable or pointer.

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );

   /* Print Book2 info */
   printBook( Book2 );

   return 0;
}

void printBook( struct Books book ) {

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

This is the output of the program

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable –

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows –

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows –

```
struct_pointer->title;
```

Example

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );

   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );

   return 0;
}

void printBook( struct Books *book ) {

   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

Output

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include –

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

C allows us to do this in a structure definition by putting :bit length after the variable. For example –

```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:4;
   unsigned int my_int:9;
} pack;
```

Here, the **packed_struct** contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit **my_int.**

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

Union

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

Defining a Union

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

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

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str –

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

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

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

union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

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

```
Memory size occupied by data: 20
```

Accessing Union Members

To access any member of a union, we use the **member access operator (.).** The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program

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

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

 int main( ) {

    union Data data;

    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");

    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);

    return 0;
_}
```

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

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

Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions –

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

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

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

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

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

User Defined Data Types

C allows the feature called **`type definition`** which allows programmers to define their identifier that would represent an existing data type. There are three such types:

Data Types	Description
Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. “struct” keyword is used to define a structure.
Union	These allow storing various data types in the same memory location. Programmers can define a union with different members, but only a single member can contain a value at a given time. It is used for
Enum	Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

Input Output Files

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

Opening Files

You can use fopen() function to create a new file or to open an existing file. This call will initialize an object of the type FILE, which contains all the information necessary to control the stream. The prototype of this function call is as follows –

```
FILE *fopen( const char * filename, const char * mode );
```

Here, filename is a string literal, which you will use to name your file, and access mode can have one of the following values –

Sr. No	Mode & Description
1	**r** Opens an existing text file for reading purpose.
2	**w** Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
3	**a** Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
4	**r+** Opens a text file for both reading and writing.
5	**w+** Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start form the beginning but writing can only be appended.
6	**a+** Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones –

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Closing a file

To close a file, use the fclose( ) function. The prototype of this function is –

```
int fclose( FILE *fp );
```

The fclose(-) function returns zero on success, or EOF if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file stdio.h.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

Writing a File

Following is the simplest function to write individual characters to a stream –

```
int fputc( int c, FILE *fp );
```

The function fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error. You can use the following functions to write a null-terminated string to a stream –

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example.

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file test.txt in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

Reading File

Given below is the simplest function to read a single character from a file –

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream –

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

```
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result –

```
1 : This
2: is testing for fprintf...

3: This is testing for fputs...
```

Let's see a little more in detail about what happened here. First, fscanf() read just This because after that, it encountered a space, second call is for fgets() which reads the remaining line till it encountered end of line. Finally, the last call fgets() reads the second line completely.

Binary I/O Functions

There are two functions, that can be used for binary input and output –

```
size_t fread(void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.