UNIT I C PROGRAMMING FUNDAMENTALS- A REVIEW

Conditional statements – Control statements – Functions – Arrays – Preprocessor - Pointers - Variation in pointer declarations – Function Pointers – Function with Variable number of arguments.

UNIT-1

C Language Components

The four main components of C language are

1) The Character Set.

2) Tokens

3) Variables

4) Data Types

1) **The Character Set :** Character set is a set of valid characters that a language cab recognize. A character represents any letter, digit or any other sign.

- Letters - A,B,C……Z or a,b,c ……z.
- Digits - 0,1……9
- Special Symbols - ~!@#$%^&.....
- White Spaces - Blank space, horizontal tab, carriage return, new line, form feed.

2) **Tokens:** The smallest individual unit in a program is known as a token.

C has five tokens

i. Keywords

ii. Identifiers

iii. Constants

iv. Punctuations

v. Operators

i. Keywords: Keywords are reserved word in C. They have predefined meaning cannot changed. All keywords must be written in lowercase. Eg:- auto,long,char,short etc.

ii. Identifiers: - Identifiers refer to the names of variable, functions and arrays. These are user-defined names. An identifier in C can be made up of letters, digits and underscore.

Identifiers may start with either alphabets or underscore. The underscore is used to make the identifiers easy to read and mark functions or library members.

iii. Constants: - Constants in C refers to fixed values that do not change during the execution of a program. C support several types of constants.

a. Numerical Constants

- i. Integer Constant
  1. Decimal Constant
  2. Octal Constant
  3. Hexadecimal Constant
- ii. Float Constant

b. Character Constants

- i. Single Character Constant
- ii. String Constant

Integer Constant: - An integer constant is a whole number without any fractional part. C has three types of integer constants.

Decimal Constant: - Decimal integers consists of digits from 0 through 9

Eg.: 34,900,3457,-978

Octal Constant: - An Octal integer constant can be any combination of digits from 0 through 7. In C the first digit of an octal number must be a zero(0) so as to differentiate it from a decimal number. Eg.: 06,034,-07564

Hexadecimal Constant: Hexadecimal integer constants can be any combination of digits 0 through 9 and alphabets from „a‟ through „f‟ or „A‟ through „F‟. In C, a hexadecimal constant must begin with 0x or 0X (zero x) so as to differentiate it from a decimal number. Eg:- 0x50,0XAC2 etc

Floating Constants (Real): Real or floating point numbers can contain both an integer part and a fractional part in the number. Floating point numbers may be represented in two forms, either in the fractional form or in the exponent form.

A float point number in fractional form consists of signed or unsigned digits including decimal point between digits. E.g:- 18.5, .18 etc.

Very large and very small numbers are represented using the exponent form. The exponent notation use the „E‟ or „e‟ symbol in the representation of the number. The number before the „E‟ is called as mantissa and the number after forms the exponent.

Eg.:-5.3E-5,-6.79E3,78e05

Single Character Constant: - A character constant in usually a single character or any symbol enclosed by apostrophes or single quotes. Eg.: ch="a"

String Constant: - A sequence of character enclosed between double quotes is called string constant. Eg.: "Hello Good Morning"

iv) Punctuations: - 23 characters are used as punctuations in C. eg: + _ / ; : > ! etc

v) Operators: - An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation on data stored in variables. The variables that are operated as operands. C operator can be classified into 8 types.

i. Arithmetic Operators : + - * / %

ii. Assignment Operators : =

iii. Relational Operators: < > <= >= == !=

iv. Logical Operators:! && ||

v. Conditional Operators: ! :

vi. Increment & Decrement Operator : ++ --

vii. Bitwise Operator:! & | ~ ^ << >>

viii. Special Operator : sizeof ,(comma)

3) Variables: A variable is an object or element that may take on any value or a specified type. Variable are nothing but identifiers, which are used to identify variables programming elements. Eg: name, sum, stu_name, acc_no etc.

4) **Data types:** Data types indicate the types of data a variable can have. A data types usually define a set of values, which can be stored in the variable along with the operations that may be performed on those values. C includes two types of data.

1. **Simple or Fundamental data type**
2. **Derived data type**

Simple Data Types:

There are four simple data types in C.

- **int**
- **char**
- **float**
- **double**

int:- This means that the variable is an integer are stored in 2 bytes, may range from -32768 to 32767.

char:- This means that the variable is a character type, char objects are stored in one byte. If unsigned, the values may be in the range 0 to 255.

Float:- This means that the variable is a real number stored in 4 bytes or 32 bits. The range of floating point values are between 3.4E-38 to 3.4E38 or 6 significant digits after the decimal point.

Double: This means that the variable is a double precision float type. In most cases the systems allocates 8 bytes or 64 bits space, between 1.7E-308 to 1.7E308.

Derived Data Types: Derived data types are constructed from the simple data types and or other derived data types. Derived data include arrays, functions, pointers, references, constants, structures, unions and enumerations.

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Unsigned character	unsigned char	1	0 to 255
Integer	int	2	-32768 to +32767
Short integer	short int	2	-32768 to 32767
Long integer	long int	4	-2147483648 to 2147483647
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short int	2	0 to 65535
Unsigned long integer	unsigned long int	4	0 to 4294967295
Float	float	4	3.4E +/- 38 (7 digits)
Double	double	8	1.7E +/- 308 (15 digits)
Long Double	long double	10	3.4E-4932 to 1.1E+4932

Precedence of Operators:

Precedence is defined as the order in which the operators in a complex evaluation are evaluated.

Associativity:

When two or more operators have the same precedence then the concept of associativity comes into discussion.

Type Conversion

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ``narrower'' operand into a ``wider'' one without losing information, such as converting an integer into floating point in an expression like f + i. Expressions that don't make sense, like using a float as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A char is just a small integer, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function atoi, which converts a string of digits into its numeric equivalent.

Implicit arithmetic conversions work much as expected. In general, if an operator like + or * that takes two operands (a binary operator) has operands of different types, the ``lower'' type is *promoted* to the ``higher'' type before the operation proceeds. The result is of the integer type. If there are no unsigned operands, however, the following informal set of rules will suffice:

- If either operand is long double, convert the other to long double.
- Otherwise, if either operand is double, convert the other to double.
- Otherwise, if either operand is float, convert the other to float.
- Otherwise, convert char and short to int.
- Then, if either operand is long, convert the other to long.

Input-output in C:

Input: In any programming language input means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Output: In any programming language output means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data.

There are two types of I/O statements in C.They are ***unformatted I/O functions*** and ***formatted I/O functions***.

Unformatted I/O functions:

1.getchar():Used to read a character

2.putchar():Used to display a character

3.gets():Used to read a string

4.puts():Used to display a string which is passed as argument to the function

Formatted I/O functions:

printf() and scanf() are examples of formatted I/O functions.

printf() is an example of formatted output function and scanf() is an example of formatted input function.

VARIOUS CONTROL STRUCTURES IN C

IF STATEMENTS

The if statements allows branching (decision making) depending upon the value or state of variables. This allows statements to be executed or skipped, depending upon decisions. The basic format is,

```
if( expression )
program statement;
```

Example:

```
if( students < 65 )
++student_count;
```

IF ELSE

The general format for these is,

```
if( condition 1 )
```

```
statement1;
else if( condition 2 )
 statement2;
else if( condition 3 )
 statement3;
else
statement4;
```

The else clause allows action to be taken where the condition evaluates as false (zero). The following program uses an if else statement to validate the users input to be in the range 1-10.

Example:

```
#include <stdio.h>
main()
{
int number;
int valid = 0;
while( valid == 0 )
{
printf("Enter a number between 1 and 10 -->");
scanf("%d",&number);
if( number < 1 )
{
printf("Number is below 1. Please re-enter\n");
valid = 0;
}
else if( number > 10 )
{
printf("Number is above 10. Please re-enter\n");
valid = 0;
}
else
valid = 1;
}
printf("The number is %d\n", number );
```

```
}
```

NESTED IF ELSE

Example:

```
#include <stdio.h>
main()
{
int invalid_operator = 0;
char operator;
float number1, number2, result;
printf("Enter two numbers and an operator in the format\n");
printf(" number1 operator number2\n");
scanf("%f %c %f", &number1, &operator, &number2);
if(operator == '*')
result = number1 * number2;
else if(operator == '/')
result =number1 / number2;
else if(operator == '+')
result = number1 + number2;
else if(operator == '-')
result = number1 - number2;
else
invalid_operator = 1;
if( invalid_operator != 1 )
printf("%f %c %f is %f\n", number1, operator, number2, result );
else
printf("Invalid operator.\n");
```

SWITCH CASE:

The switch case statement is a better way of writing a program when a series of if elses occurs. The general format for this is,

```
switch ( expression )
```

```
{
case value 1:
program statement;
program statement;
.....
break;
case value n:
program statement;
.......
break;
default:
.......
break;
}
```

The keyword break must be included at the end of each case statement. The default clause is optional, and is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

Example:

```
#include <stdio.h>
main()
{
int menu, numb1, numb2, total;
printf("enter in two numbers -->");
scanf("%d %d", &numb1, &numb2 );
 printf("enter in choice\n");
printf("1=addition\n");
printf("2=subtraction\n");
scanf("%d", &menu );
switch( menu )
{
case 1:
total = numb1 + numb2;
break;
```

```
case 2:
total = numb1 -numb2;
break;
default:
printf("Invalid option selected\n");
}
if( menu == 1 )
printf("%d plus %d is %d\n", numb1, numb2, total );
else if( menu == 2 )
printf("%d minus %d is %d\n", numb1, numb2, total );
}
```

The above program uses a switch statement to validate and select upon the users input choice, simulating a simple menu of choices.

FOR LOOPS

The basic format of the for statement is,

```
for( start condition; continue condition; re-evaulation )
program statement;
```

Example:

```
/* sample program using a for statement*/
#include <stdio.h>
main()
{
int count;
for( count = 1; count <= 10; count = count + 1 )
printf("%d ", count );
printf("\n");
}
```

The program declares an integer variable count. The first part of the for statement count = 1;initialises the value of count to 1. The for loop continues whilst the condition count <= 10; evaluates as TRUE. As the variable count has just been

initialised to 1, this condition is TRUE and so the program statement printf("%d ", count); is executed, which prints the value of count to the screen, followed by a space character. Next, the remaining statement of the for is executed count = count + 1 , which adds one to the current value of count. Control now passes back to the conditional test, count <= 10; which evaluates as true, so the program statement printf("%d ", count); is executed. Count is incremented again, the condition re-evaluated etc, until count reaches a value of 11.

When this occurs, the conditional test count <= 10; evaluates as FALSE, and the for loop terminates, and program control passes to the statement printf("\n"); which prints a newline, and then the program terminates, as there are no more statements left to execute.

THE WHILE STATEMENT

The while provides a mechanism for repeating C statements whilst a condition is true. Its format is,

while(condition)

program statement;

Somewhere within the body of the while loop a statement must alter the value of the condition to allow the loop to finish.

Example:

```
#include <stdio.h>
main()
{
int loop = 0;
while( loop <= 10 )
{
printf("%d\n", loop);
++loop;
}
}
```

The above program uses a while loop to repeat the statements

```
printf("%d\n", loop);
++loop;
```

while the value of the variable loop is less than or equal to 10.

Note how the variable upon which the while is dependant is initialised prior to the while statement (in this case the previous line), and also that the value of the variable is altered within the loop, so that eventually the conditional test will succeed and the while loop will terminate. This program is functionally equivalent to the earlier for program which counted to ten.

THE DO WHILE STATEMENT

The do { } while statement allows a loop to continue whilst a condition evaluates as TRUE (non-zero).

The loop is executed as least once.

Example:

```
#include<stdio.h>
main()
{
int value, r_digit;
printf("Enter the number to be reversed.\n");
scanf("%d", &value);
do
{
r_digit = value % 10;
printf("%d", r_digit); value = value / 10;
}
while( value !=0 );
printf("\n");
}
```

The above program reverses a number that is entered by the user. It does this by using the modulus % operator to extract the right most digits into the variable r_digit. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0.

BREAK AND CONTINUE STATEMENTS

C break statement is used to terminate any type of loop such as while loop, do while loop and for loop. C break statement terminates the loop body immediately and passes control to the next statement after the loop.

C continue statement is used to skip over the rest of the current iteration. After continue statement, the control returns to the top of the loop.

Example program for continue:

```
main()
{
int i;
int j = 10;
for( i = 0; i <= j; i ++ )
{
if( i == 5 )
{
continue;
}

printf("Hello %d\n", i );
}
}
```

OUTPUT:

Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10

GOTO STATEMENT:

A goto statement is used to branch (transfer control) to another location in a program.

Syntax:

goto label;

The label can be any valid identifier name and it should be included in the program followed by a colon.

Storage Classes

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

Scope: The scope of variable determines over what region of the program a variable is actually available for use.

Visibility: The program‟s ability to access a variable from the memory.

Lifetime: Life time refers to the period during which a variable retains a given value during the execution of a program.

Scope rules:

1. The scope of a global variable is the entire program file.

2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.\

3. The scope of a formal function argument is its own function.

4. The life time of an auto variable declared in main is the entire program execution time, although its scope is only the main function.

5. The life of an auto variable declared in a function ends when the function is exited.

6. All variables have visibility in their scope , provided they are not declared again.

7. A variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

These are following storage classes which can be used in a C Program:

- auto
- register

- static
- extern

auto - Storage Class

auto is the default storage class for all local variables and the local variable is known only to the function in which it is declared. If the value is not assigned to the variable of type auto the default value is garbage value.

Syntax :

auto [data_type] [variable_name];

Example :

auto int a;

Program :

```
/* Program to demonstrate automatic storage class.*/
#include <stdio.h>
#include <conio.h>
void main()
{
auto int i=10;
clrscr();
{
auto int i=20;
printf("\n\t %d",i);
}
printf("\n\n\t %d",i);

getch();
}
```

Output :

20

10

register - Storage Class

Register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location). If the value is not assigned to the variable of type register the default value is garbage value.

Syntax :

register [data_type] [variable_name];

Example :

register int a;

When the calculations are done in CPU, the value of variables are transferred from main memory to CPU. Calculations are done and the final result is sent back to main memory. This leads to slowing down of processes. Register variables occur in CPU and value of that register variable is stored in a register within that CPU. Thus, it increases the resultant speed of operations. There is no waste of time, getting variables from memory and sending it to back again.It is not applicable for arrays, structures or pointers.It cannot be used with static or external storage class.

Program :

```
/* Program to demonstrate register storage class.*/
#include <stdio.h>
#include <conio.h>
void main()
{
register int i=10;
clrscr();
{
register int i=20;
printf("\n\t %d",i);
}
printf("\n\n\t %d",i);
getch();
}
```

Output :

20
10

static - Storage Class

static is the default storage class for global variables. As the name suggests the value of static variables persists until the end of the program.Static can also be defined within a function. If this is done the variable is initalised at run time but is not reinitalized when the function is called. This inside a function static variable retains its value during vairous calls. If the value is not assigned to the variable of type static the default value is zero.

Syntax :

static [data_type] [variable_name];

Example :

static int a;

There are two types of static variables: a) Local Static Variable b) Global Static Variable

Static storage class can be used only if we want the value of a variable to persist between different function calls.

Program :

```
/* Program to demonstrate static storage class. */
#include <stdio.h>
#include <conio.h>
void main()
{
int i;
void incre(void);
clrscr();
for (i=0; i<3; i++)
incre();
getch();
}
void incre(void)
{
int j=1;
static int k=1;
```

```
j++;
k++;

printf("\n\n Automatic variable value : %d",j);
printf("\t Static variable value : %d",k);
}
```

Output :

Automatic variable value : 2 Static variable value : 2

Automatic variable value : 2 Static variable value : 3

Automatic variable value : 2 Static variable value : 4

extern - Storage Class

Variables that are both alive and active throughout the entire program are known as external variables. **extern** is used to give a reference of a global variable that is visible to all the program files. If the value is not assigned to the variable of type static the default value is zero.

Syntax :

extern [data_type] [variable_name];

Example :

extern int a;

The variables of this class can be referred to as 'global or external variables.' They are declared outside the functions and can be invoked at anywhere in a program.

Program :

```
/* Program to demonstrate external storage class.*/
#include <stdio.h>
#include <conio.h>
extern int i=10;
void main()
{
int i=20;
void show(void);
clrscr();
printf("\n\t %d",i);
```

```
show();
getch();
}
void show(void)
{
printf("\n\n\t %d",i);
}
```

Output :

20

10

FUNCTIONS

A function is a group of statements that together perform a task. C enables its programmer to break-up a program into segment commonly known as function, each of which can be written more or less independently of the others. Every function in the program supposed to perform a well defined task. It is needed because, dividing the program into separate well defined functions facilitates each function to be written and tested separately. Understanding, coding and testing multiple separate function are easier than doing the same for one huge function.

Types:

1. Built in Functions(Library functions)
2. User Defined functions

Library functions:

C standard library provides a rich collection of functions for performing I/O operations, mathematical calculations, string manipulation operations etc.For example, ***sqrt(x)*** is a function to calculate the square root of a number provided by the C standard library and included in the <math.h>header file.

Note that each program in C has a function called ***main*** which is used as the root function of calling other library functions.

Programmer Defined functions:

In C, the programmers can write their own functions and use them in their programs.

The user defined function can be explained with three elements.

1. Function definition
2. Function call
3. 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 function call. The program that calls the function is known as 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 the function prototype.

Structure of a function:

There are two main parts of the function. The function header and the function body.

Consider the following example:

```
int sum(int x, int y)
{
int ans = 0;
ans = x + y;
return ans;
}
```

Function Header

In the first line of the above code

int sum(int x, int y)

It has three main parts

1. The name of the function i.e. *sum*
2. The parameters of the function enclosed in paranthesis
3. Return value type i.e. *int*

Function Body

Whatever is written with in { } in the above example is the body of the function.

Function prototype: Function prototypes are always declared at the beginning of the program indicating the name of the function, the data type of its arguments which is passed to the function and the data type of the returned value from the function.

Ex: int square(int);

The prototype of a function provides the basic information about a function which tells the compiler that the function is used correctly or not. It contains the same information as the function header contains. The prototype of the function in the above example would be like

int sum (int x, int y);

The only difference between the header and the prototype is the semicolon ; there must the a semicolon at the end of the prototype.

Categories of functions:

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

1. Functions with no arguments and no return values
2. Functions with arguments and no return values
3. Fucntions with arguments and a return value
4. Functions with no arguments but a return value

Example programs:

/* The program illustrates the **functions with no arguments** and no return value*/

```
#include<stdio.h>
void add(void);
void main()
{
add();
}
void add(void)
{
int x,y,sum;
printf("Enter any two integers:");
scanf("%d%d",&x,&y);
sum=x+y;
printf("The sum id %d",sum);
}
```

Output:

Enter any two integers: 2 4

The sum is 6

/* The program illustrates the **functions with arguments** and no return value*/

```
#include<stdio.h>
void add(int a,int b);
void main()
{
int x,y,;
printf("Enter any two integers:");
scanf("%d%d",&x,&y);
add(x,y);
}
void add(int a,int b)
{
int sum;
sum=a+b;
printf("The sum id %d",sum);
}
```

Output:

Enter any two integers: 2 4

The sum is 6

/* The program illustrates the **functions with arguments and a return value***/

```
#include<stdio.h>
int add(int a,int b);
void main()
{
int x,y,sum;
printf("Enter any two integers:");
scanf("%d%d",&x,&y);
sum=add(x,y);
printf("The sum id %d",sum);
}
```

```
int add(int a,int b)
{
int c;
c=a+b;
return c;
}
```

Output:

Enter any two integers: 2 4

The sum is 6

/* The program illustrates the **functions with no arguments and but a return value***/

```
#include<stdio.h>
int add(void);
void main()
{
int sum;
sum=add();
printf("The sum id %d",sum);
}
void add(void)
{
int x,y,c;
printf("Enter any two integers:");
scanf("%d%d",&x,&y);
c=x+y;
return c;
}
```

Output: Enter any two integers: 2 4

The sum is 6

Types of function calls

Call by Value:

When a function is called by an argument/parameter the copy of ***the argument*** is passed to the function. If the argument is a normal (non-pointer) value a possible change on the copy of the argument in the function does not change the original value of the argument. However, given a pointer/address value any change to the value pointed by the pointer/address changes the original argument.

Example 1: The following program is to calculate and print the area and the perimeter of a circle. by using ***Call by value*** approach

```
#include<stdio.h>/
#define pi 3.14
float area(float);
float perimeter(float);
int main( )
{
float r, a, p;
printf("Enter the radius\n");
scanf("%f",&r);
a = area(r);
p = perimeter(r);
printf("The area = %.2f, \n The Perimeter = %.2f", a, p);
return 0;
}

float area(float x)
{
return pi*r*r;
}
float perimeter(float y)
{
return 2.0*pi*r;
}
```

Example 2:

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

```
void add( int n);
int main()
{
        int num = 2;
        printf("\n The value of num before calling the function = %d", num);
        add(num);
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add(int n)
{
        n = n + 10;
        printf("\n The value of num in the called function = %d", n);
}
```

The output of this program is:

The value of num before calling the function = 2

The value of num in the called function = 20

The value of num after calling the function = 2

Call by Reference: When a function is called by an argument/parameter which is a pointer(address of the argument) the copy of ***the address of the argument*** is passed to the function. Therefore a possible change on the data at the referenced address change the original value of the argument.

Example 1: The following program is to calculate and print the area and the perimeter of a circle. by using ***Call by reference*** approach

```
#include<stdio.h>
#define pi 3.14
void area_perimeter(float, float *, float *);
int main( )
{
float r, a, p;
```

```
printf(“Enter the readius\n”);
scanf(“%f”,&r);
area_perimeter(r,&a,&p);
printf(“The area = %.2f, \n The Perimeter = %.2f”, a, p);
return 0;
}
void area_perimeter(float x, float *aptr, float *pptr);
{
*aptr = pi*x*x;
*pptr = 2.0*pi*x;
}
```

Example 2:

```
#include<stdio.h>
void add( int *);
0int main()
{
        int num = 2;
        printf("\n The value of num before calling the function = %d", num);
        add(&num);
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add( int *n)
{
        *n =* n + 10;
        printf("\n The value of num in the called function = %d", *n);
}
```

The output of this program is:

The value of num before calling the function = 2

The value of num in the called function = 20
The value of num after calling the function = 20

RECURSION

Recursion:

Recursion is a process in which a function calls itself.

Example:

```
Recursion()
{
printf("Recursion !");
Recursion();
}
```

- A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases, they are

  base case, in which the problem is simple enough to be solved directly without making any further calls to the same function

  recursive case, in which first the problem at hand is divided into simpler sub parts. Second the function calls itself but with sub parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
- Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem. In recursive function, complicated problem is defined in terms of simpler problems and the simplest problem is given explicitly.

TYPES OF RECURSION

Any recursive function can be characterized based on:

1. whether the function calls itself directly or indirectly (direct or indirect recursion).
2. whether any operation is pending at each recursive call (tail-recursive or not).

3. the structure of the calling pattern (linear or tree-recursive).

DIRECT RECURSION

A function is said to be *directly* recursive if it explicitly calls itself. For example, consider the function given below.

```
int Func( int n)
{
        if(n==0)
                retrun n;
        return (Func(n-1));
}
```

INDIRECT RECURSION

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
                return n;
        return Func2(n);
}
```

```
int Func2(int x)
{
        return Func1(x-1);
}
```

TAIL RECURSION

- A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller.
- That is, when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(n)
{
  return Fact1(n, 1);
}
```

```
int Fact1(int n, int res)
{
        if (n==1)
           return res;
        return Fact1(n-1, n*res);
}
```

LINEAR AND TREE RECURSION

- Recursive functions can also be characterized depending on the way in which the recursion grows- in a linear fashion or forming a tree structure.
- In simple words, a recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function. For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to Fact.
- On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.

Example Program for recursion(Factorial) :

```
/* Program to demonstrate recursion */.
#include <stdio.h>
int fact(int x);
void main()
```

```
{
int a,f;
clrscr();
printf("Enter any integer:");
scanf("%d",&a);
f=fact(a);
printf("The factorial of %d is %d",a,f);
}
int fact(int x)
{
if(x==0)
return 1;
else
return(x*fact(x-1));
}
```

Output:

Enter any integer:5

The factorial of 5 is 120

Features :

· There should be at least one if statement used to terminate recursion.

· It does not contain any looping statements.

Advantages :

· It is easy to use.

· It represents compact programming structures.

Disadvantages :

It is slower than that of looping statements because each time function is called.

Example:

Fibonacci series using recursion

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int);
main()
{
int n;
printf("\n enter the number of terms in the series:");
scanf("%d",&n);
for(i=0;i<n;i++)
printf("\n fibonacci (%d)=%d",i,fibonacci(i));
return 0;
}
int fibonacci(int num)
{
if(num<=2)
return 1;
else
return(fibonacci(num-1)+fibonacci(num-2));
}
```

ARRAYS

Definition:

An array is a derived data type, which is a collection of element of same type stored in a physically continues memory location. Example, int marks [10];

Types of arrays:

1) Single dimensional array. eg: int a[5];
2) Multi dimensional array. eg: int a[5] [5];

List of Operations performed on array:

- Insertion: insert an element into the array.
- Deletion: Delete an element from the array.
- Traversal: Accesses each element of the array.
- Sort: Re-arranges the array element in a specific order.
- Search: Searches the key value in the array.

Calculating the length of the Array:

The length of the array is given by the number of elements stored in it. The general formula to calculate the length the array is, length = index of the last element – index of the first element +1.

Declaration of an Array

Arrays must be declared before they can be used in the program. Standard array declaration is as:

type variable_name[lengthofarray];

Here type specifies the variable type of the element which is going to be stored in the array.

Example:

double height[10];

float width[20];

In C Language, array starts at position 0. The elements of the array occupy adjacent locations in memory. C Language treats the name of the array as if it was a pointer to the first element This is important in understanding how to do arithmetic with arrays. Any item in the

array can be accessed through its index, and it can be accesed any where from with in the program. So,

m=height[0];

variable m will have the value of first item of array height.

Initializing Arrays

The following is an example which declares and initializes an array of nine elements of type int. Array can also be initialized after declaration.

int scores[9]={23,45,12,67,95,45,56,34,83};

Two dimensional arrays

C allows us to define the table of items by using two dimensional arrays.

Declaration:

Two dimensional arrays are declared as follows:

type array_name[row_size][column_size];

Initialization

Like one dimensional arrays, two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Ex: int table[2][3]={0,0,0,1,1,1};

Initializes the elements of first row to zero and second row to one. The initialization is also done row by row. The above statement can be equivalently written as

Ex: int table[2][3]={0,0,0},{1,1,1}};

The following is a sample program that stores roll numbers and marks obtained by a student side by side in matrix,

```
main ( )
{
int stud [4] [2];
int i, j;
for (i =0; i < =3; i ++)
{
printf ("\n Enter roll no. and marks");
scanf ("%d%d", &stud [i] [0], &stud [i] [1] );
}
for (i = 0; i < = 3; i ++)
```

```
printf ("\n %d %d", stud [i] [0], stud [i] [1]):
}
```

Multidimensional Arrays

C allows arrays of three or more dimensions. The general form of multidimensional array is

type array_name[s1][s2]…..[s3];

where si is the size of ith dimension..

Ex: int survey[3][5][12];

survey is a 3 dimensional array.

Passing arrays through functions:

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts and the size of the array as arguments.

Ex:largest(a,n);

will pass all the elements contained in the array a of size n. The called function expecting this call must be appropriately defined.

Example program:

```
main()
{
float largest();
static float value[4]={10,-4,2,-3};
printf(“%f”,largest(value,4));
}
float largest(a,n)
float a[];
int n;
{
int i;
float max;
max=a[0];
for(i=1;i<n;i++)
```

if(max<a[i])

max=a[i];

return(max);

}

To process arrays in a large program, we have to be able to pass them to functions. We can pass arrays in two ways: pass individual elements or pass the whole array.

Array of Strings

Consider, for example, the need to store the days of the week in their textual format. We could create a two-dimensional array of seven days by ten characters, but this wastes space

Program to print Days of the Week

```
/* Demonstrates an array of pointers to strings.
      Written by:
      Date written:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char*  pDays[7];
   char** pLast;

// Statements
   pDays[0] = "Sunday";
   pDays[1] = "Monday";
   pDays[2] = "Tuesday";
   pDays[3] = "Wednesday";
   pDays[4] = "Thursday";
   pDays[5] = "Friday";
   pDays[6] = "Saturday";

   printf("The days of the week\n");
   pLast = pDays + 6;
   for (char** pWalker = pDays;
               pWalker <= pLast;
               pWalker++)
        printf("%s\n", *pWalker);
   return 0;
} // main
```

Example programs:

Program to insert a number at a given location in an array.

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,n,num,pos,arr[10];
clrscr();
printf("\n enter the number of elements in the array:");
scanf("%d",&arr[i]);
for(i=0;i<n;i++)
{
printf("\n arr[%d]=:",i)
scanf("%d",&arr[i]);
}
printf("\n enter the number to be inserted:");
scanf(""%d",&num);
printf("enter the position at which the number has to be added:");
scanf("%d",&pos);
for(i=n;i>=pos;i--)
{
arr[i+1]=arr[i];
arr[pos]=num;
printf("\n the array after insertion of %d is:",num);
```

```
for(i=0;i<n+1;i++)
{
printf("\n arr[%d]=%d",i,arr[i]);
getch();
}
```

Program to merge two unsorted Arrays.

Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence,the merged array contains the contents of the first array followed by the contents of the second array.

```
#include<stdio.h>
#include<conio.h>
main()
{
int arr1[10],arr2[10],arr3[20];
int i,n1,n2,m,index=0;
clrscr();
printf("\n printf("\n enter the number of elements in the array1:");
scanf("%d",&n1);
printf("\n enter the elements of the array1:");
for(i=0;i<n1;i++)
{
printf("\n arr1[%d]=",i);
```

```
scanf("%d",arr1[i]);
}
printf("\n enter the number of elements in the array2:");
scanf("%d",&n2);
printf("\n enter the elements of the array2:");
for(i=0;i<n2;i++)
{
printf("\n arr2[%d]=",i);
scanf("%d",arr2[i]);
}
m=n1+n2;
for(i=0;i<n1;i++)
{
arr3[index]=arr1[i];
index++;
}
for(i=0;i<n2;i++)
{
arr3[index]=arr2[i];
index++;
}
printf("\n\n the merged array is");
for(i=0;i<m;i++)
```

```
printf("\n arr[%d]=%d",i,arr3[i]);

getch();

}
```

VARIOUS PRE-PROCESSOR DIRECTIVES

- The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of preprocessor directive which is placed in the source program before the main().
- Before the source code is passed through the compiler, it is examined by the preprocessor for any preprocessor directives. In case, the program has some preprocessor directives, appropriate actions are taken (and the source program is handed over to the compiler.
- The preprocessor directives are always preceded by a hash sign (#).
- The preprocessor is executed before the actual compilation of program code begins. Therefore, the preprocessor expands all the directives and take the corresponding actions before any code is generated by the program statements.
- No semicolon (;) can be placed at the end of a preprocessor directive.

The advantages of using preprocessor directives in a C program include:

- Program becomes readable and easy to understand
- Program can be easily modified or updated
- Program becomes portable as preprocessor directives makes it easy to compile the program in different execution environments
- Due to the aforesaid reason the program also becomes more efficient to use.

TYPES OF PREPROCESSOR DIRECTIVES

1. Macro replacement directive (#define,#undef)
2. Source file inclusion directive(#include)
3. Line directive(#line)

4. Error directive(#error)
5. Pragma directive(#pragma)
6. Conditional compilation directives(#if,#else,#elif,#endif,#ifdef,#ifndef)

1. MACRO REPLACEMENT DIRECTIVE (#define,#undef)

A Macro is a facility provided by the C preprocessor, by which a token can be replaced by the User-defined sequence of characters. Macro name are generally written in upper case.

#undef

As the name suggests, the #undef directive undefines or removes a macro name previously created with #define. Undefining a macro means to cancel its definition. This is done by writing #undef followed by the macro name that has to be undefined.

#define

- To define preprocessor macros we use #define. The #define statement is also known as macro definition or simply a macro. There are two types of macros- object like macro and function like macro.

Types of Macro:

1. Macro without Arguments, also called as Object-like macros.

2. Macro with Arguments, also called as Function-like macros.

Object like macro

- An *object-like macro* is a simple identifier which will be replaced by a code fragment. They are usually used to give symbolic names to numeric constants. Object like macros do not take any argument. It is the same what we have been using to declare constants using #define directive. The general syntax of defining a macro can be given as:

  #define identifier string

- The preprocessor replaces every occurrence of the identifier in the source code by a string.

Example

```
#include<stdio.h>
 #define PI 3.14
main()
{
int rad=5;
Printf("area of circle is %f",PI*rad*rad);
}
```

Function-like macros

- They are used to stimulate functions.
- When a function is stimulated using a macro, the macro definition replaces the function definition.
- The name of the macro serves as the header and the macro body serves as the function body. The name of the macro will then be used to replace the function call.
- **The function-like macro includes a list of parameters.**
- References to such macros look like function calls. However, when a macro is referenced, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments, and the text is inserted into the program stream. Therefore, macros are considered to be much more efficient than functions as they avoid the overhead involved in calling a function.
- The **syntax** of defining a function like macro can be given as

```
# define identifier(arg1,arg2,...argn) string
```

Example

```
#include<stdio.h>
#include<conio.h>
#define s(x) x*x
Void main()
{
Clrscr();
```

```
Printf("%d",s(5));
}
```

Output:

25

Macros and Function:

Macros are more efficient (and faster) than function, because their corresponding code is inserted directly at the point where the macro is called. There is no overhead involved in using a macro like there is in placing a call to a function.

However, macros are generally small and cannot handle large, complex coding constructs. In cases where large, complex constructs are to handled, functions are more suited, additionally; macros are expanded inline, which means that the code is replicated for each occurrence of a macro.

2. SOURCE FILE INCLUSION DIRECTIVE

#include

- An external file containing function, variables or macro definitions can be included as a part of our program. This avoids the effort to re-write the code that is already written.
- The #include directive is used to inform the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.
- The #include directive can be used in two forms. Both forms makes the preprocessor insert the entire contents of the specified file into the source code of our program. However, the difference between the two is the way in which they search for the specified.

  #include <*filename*>

- This variant is used for system header files. When we include a file using angular brackets, a search is made for the file named *filename* in a standard list of system directories.

  #include "*filename*"
- This variant is used for header files of your own program. When we include a file using double quotes, the preprocessor searches the file named *filename* first in the directory containing the current file, then in the quote directories and then the same directories used for <*filename*>.

Example:

File name 1: T.cpp

```
#include<stdio.h>
#include<conio.h>
#include “max.cpp”
void main()
{
clrscr();
printf(“%d”,max());
}
```

File name 2: max.cpp

```
Void max()
{
Int a=5,b=3,m;
m=(a>b)?a:b;
Printf(“%d”,m);
}
```

3. LINE DIRECTIVE

The line directive is used to reset the line number. The line directive is used for the purpose of error diagnostics. It has two forms:

1. #line constant
2. #line constant “filename”

#line

- Compile the following C program

```
#include<stdio.h>
main()
{       int a=10:
        printf("%d", a);
}
```

- The above program has a compile time error because instead of a semi-colon there is a colon that ends line int a = 10. So when you compile this program an error is generated during the compiling process and the compiler will show an error message with references to the name of the file where the error happened and a line number. This makes it easy to detect the erroneous code and rectify it.
- The #line directive enables the users to control the line numbers within the code files as well as the file name that we want that appears when an error takes place. The syntax of #line directive is:
- #line line_number filename
- Here, line_number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.
- Filename is an optional parameter that redefines the file name that will appear in case an error occurs. The filename must be enclosed within double quotes. If no filename is specified then the compiler will show the original file name. For example:

```
#include<stdio.h>
main()
{       #line 10 "Error.C"
        int a=10:
        #line 20
        printf("%d, a);
}
```

4. PRAGMA DIRECTIVES

The #pragma directive is a compiler-specific directive, used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. The effect of pragma will be applied from the point where it is included to the end of the compilation unit or until another pragma changes its status. A #pragma directive is an instruction to the compiler and is usually ignored during preprocessing.

- The #pragma directive is used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.
- The effect of pragma will be applied from the point where it is included to the end of the compilation unit or until another pragma changes its status.
- A #pragma directive is an instruction to the compiler and is usually ignored during preprocessing.
- The syntax of using a pragma directive can be given as:

  #pragma *string*

  The commonly used forms of the pragma directives are:

  1. #pragma option (-C,-C-,-G,-G-,-r,-r-,-a,-a-)
  2. #pragma warn (dup,voi,rvl,par,pia,rch,aus)
  3. #pragma startup and #pragma exit

Example:

```
#include<stdio.h>
#include<conio.h>
#pragma warn –rvl
main()
{
Printf(“hi”);
}
```

Output: **hi**

Note: Without the pragma directive, the output is, **Function should return a value.**

5. **CONDITIONAL DIRECTIVES**

- A conditional directive is used instruct the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. The preprocessor conditional directives can test arithmetic expressions, or whether a name is defined as a macro, etc.
- Conditional preprocessor directives can be used in the following situations:
- A program may need to use different code depending on the machine or operating system it is to run on.
- The conditional preprocessor directive is very useful when you want to compile the same source file into two different programs. While one program might make frequent time-consuming consistency checks on its intermediate data, or print the values of those data for debugging, the other program, on the other hand can avoid such checks.
- The conditional preprocessor directives can be used to exclude code from the program whose condition is always false but is needed to keep it as a sort of comment for future reference.

#ifdef

- #ifdef is the simplest sort of conditional preprocessor directive and is used to check for the existence of macro definitions.
- Its syntax can be given as:

```
#ifdef MACRO
        controlled text
#endif
#ifdef MAX
int STACK[MAX];
#endif
```

#ifndef

The #ifndef directive is the opposite of #ifdef directive. It checks whether the MACRO has not been defined or if its definition has been removed with #undef.

#ifndef is successful and returns a non-zero value if the MACRO has not been defined. Otherwise in case of failure, that is when the MACRO has already been defined, #ifndef returns false (0).

The general format to use #ifndef is the same as for #ifdef:

#ifndef *MACRO*

controlled text

#endif

The #if Directive

- The #if directive is used to control the compilation of portions of a source file. If the specified condition (after the #if) has a nonzero value, the controlled text immediately following the #if directive is retained in the translation unit.
- The #if directive in its simplest form consists of

#if condition

controlled text

#endif

- While using #if directive, make sure that each #if directive must be matched by a closing #endif directive. Any number of #elif directives can appear between the #if and #endif directives, but at most one #else directive is allowed. However, the #else directive (if present) must be the last directive before #endif.

The #else Directive

- The #else directive can be used within the controlled text of an #if directive to provide alternative text to be used if the condition is false.
- The general format of #else directive can be given as:

#if condition

Controlled text1

#else

Controlled text2

#endif

The #elif Directive

- The #elif directive is used when there are more than two possible alternatives. The #elif directive like the #else directive is embedded within the #if directive and has the following syntax:

#if condition
Controlled text1
#elif new_condition
Controlled text2
#else
Controlled text3
#endif

The #endif Directive

- The general syntax of #endif preprocessor directive which is used to end the conditional compilation directive can be given as:

#endif

- The #endif directive ends the scope of the #if , #ifdef , #ifndef , #else , or #elif directives.

6. #error DIRECTIVE

- The #error directive is used to produce compiler-time error messages. The syntax of this directive is:
  #error string
- The error messages include the argument *string.* The #error directive is usually used to detect programmer inconsistencies and violation of constraints during preprocessing.
- When #error directive is encountered, the compilation process terminates and the message specified in string is printed to stderr.
  Example:
  #ifndef SQUARE
  #error MACRO not defined.

```
#endif
#ifndef VERSION
#error Version number is not specified.
#endif
```

POINTER

Pointer is a variable which holds the address of another variable.

Pointer Declaration:

```
datatype *variable-name;
```

Example:

```
int *x, c=5;
x=&a;
```

Uses of Pointers

Pointers are used to return more than one value to the function

· Pointers are more efficient in handling the data in arrays

· Pointers reduce the length and complexity of the program

· They increase the execution speed

· The pointers save data storage space in memory

Types of Pointers

Null Pointer: It is a special pointer value that is known not to point anywhere, this menas that the null pointer does not to point to any valid memory address.

Eg: int * ptr=NULL;

Generic pointer: It is a pointer variable that has void as its data type. The void pointer, or the generic pointer, is a special tyoe of pointer that can be pointed at variable of any data type.

For example, void * ptr;

Constant Pointers

These type of pointers are the one which cannot change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). If we try to point the pointer to some other variable (or try to make the pointer store address of some other variable), then constant pointers are incapable of this.

A constant pointer is declared as : 'int *const ptr' (the location of 'const' make the pointer 'ptr' as constant pointer).

Example:

```
#include<stdio.h>
int main(void)
{
   int a[] = {10,11};
   int* const ptr = a;
   *ptr = 11;
   printf("\n value at ptr is  : %d\n",*ptr);
   printf("\n Address pointed by ptr  : %p\n",(unsigned int*)ptr);
   ptr++;
   printf("\n Address pointed by ptr  : %p\n",(unsigned int*)ptr);
   return 0;

  }
```

Pointer to Constant

These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.

A pointer to a constant is declared as : 'const int *ptr' (the location of 'const' makes the pointer 'ptr' as a pointer to constant.

Example:

```
#include<stdio.h>
int main(void)
{
   int a = 10;
   const int* ptr = &a;
   printf("\n value at ptr is  : %d\n",*ptr);
   printf("\n Address pointed by ptr  : %p\n",(unsigned int*)ptr);
   *ptr = 11;
   return 0;
}
```

ARRAY OF POINTERS

An array of pointers is a collection of addresses. The addresses in an array of pointers could be the addresses of isolated variables or the addresses of array elements or any other addresses. The only constraint is that all the pointers in an array must be of the same type.

Example:

```
#include<stdio.h>

main()

{

int a=10,b=20,c=30;

int *arr[3]={&a,&b,&c};

printf(“the values of variables are:\n”);

printf(“%d%d%d\n”,a,b,c);

printf(“%d%d%d\n”,*arr[0],*arr[1],*arr[2]);

}
```

POINTERS TO A POINTERS

You can use pointers that point to pointers. The pointers in turn, point to data (or even to other pointers). To declare pointers to pointers just add an asterisk (*) for each level of reference.

For example, if we have:

```
int x=10;
int  *px, **ppx;
px=&x;
ppx=&px;
```

Now if we write,

```
printf("\n %d", **ppx);
```

Then it would print 10, the value of x.

POINTER TO AN ARRAY

It is possible to create a pointer that points to a complete array instead of pointing to the individual elements of an array or isolated variables. Such pointer is known as pointer to an array.

Example:

```
#include<stdio.h>

main()

{

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

int (*ptr)[2]=arr;

printf("%d\n",arr[0][0]);

printf("%p\n",arr[0]);

printf("%p\n",ptr+1);
```

}

Output:

2

234F:2234

234F:2238

POINTER ARITHMETIC

Pointers can be added and subtracted. Addition and subtraction are mainly for moving forward and backward in an array.

Operator	Result
++	Goes to the next memory location that the pointer is pointing to.
--	Goes to the previous memory location that the pointer is pointing to.
-= or -	Subtracts value from pointer.
+= or +	Adding to the pointer

CHARACTER POINTER

The array declaration "char a[6];" requests that space for six characters be set aside, to be known by the name "a." That is, there is a location named "a" at which six characters can sit. The pointer declaration "char *p;" on the other hand, requests a place which holds a pointer. The pointer is to be known by the name "p," and can point to any char (or contiguous array of chars) anywhere.

The statements ,

char a[] = "hello";

char *p = "world";

would result in data structures which could be represented like this:

a: | h | e | l | l | o |

p: | *======> | w | o | r | l | d |\0 |

It is important to realize that a reference like x[3] generates different code depending on whether x is an array or a pointer. Given the declarations above, when the compiler sees the expression a[3], it emits code to start at the location "a," move three past it, and fetch the character there. When it sees the expression p[3], it emits code to start at the location "p," fetch the pointer value there, add three to the pointer, and finally fetch the character pointed to. In the example above, both a[3] and p[3] happen to be the character 'l', but the compiler gets there differently.

Example:

```
#include <stdio.h>
int main()
{
char a='b';
char *ptr;
printf("%cn",a);
ptr=&a;
printf("%pn",ptr);
*ptr='d';
printf("%cn",a);
return 0;
}
```

Output :

b

001423

d

POINTERS AND 2 DIMENSIONAL ARRAYS

Example:

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

```
char board[3][3] = {
{'1','2','3'},
{'4','5','6'},
{'7','8','9'}
};
int i;
for(i = 0; i < 9; i++)
printf(" board: %c\n", *(*board + i));
return 0;
}
```

Output:

```
board: 1
board: 2
board: 3
board: 4
board: 5
board: 6
board: 7
board: 8
board: 9
```

POINTERS AND FUNCTIONS

Pointers can be used with functions. The main use of pointers is „call by reference" functions. Call by reference function is a type of function that has pointer/s (reference) as parameters to that function. All calculation of that function will be directly performed on referred variables.

Example

```
#include <stdio.h>
 void DoubleIt(int *num)
 {
 *num*=2;
 }
 int main()
```

```
{
int number=2;
DoubleIt(&number);
printf("%d",number);
return 0;
}
```

Output

4

POINTERS AND STRUCTURES

Pointers and structures is broad topic and it can be very complex However pointers and structures are great combinations; linked lists, stacks, queues and etc are all developed using pointers and structures in advanced systems.

Example:

```
#include <stdio.h>
 struct details {
int num;
};
 int main()
{
 struct details MainDetails;
struct details *structptr;
structptr=&MainDetails;
structptr->num=20;
printf("n%d",MainDetails.num);
return 0;
}
```

Output :20

FUNCTION POINTER

POINTER TO FUNCTION (Function pointer)

- This is a useful technique for passing a function as an argument to another function.
- In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name.

```
/* pointer to function returning  int */
int (*func)(int a, float b);
```

- If we have declared a pointer to the function, then that pointer can be assigned to the address of the right sort of function just by using its name.

CALLING A FUNCTION USING FUNCTION POINTER

A function pointer can be used to call a function in any of the following two ways:

1. By explicitly dereferencing it using the dereference operator (*).
2. By using its name instead of the function's name

When a pointer to a function is declared, it can be **called** using one of two forms:

```
(*func)(1,2);   OR    func(1,2);
```

Example:

```
#include <stdio.h>
void print(int n);
main()
{
        void (*fp)(int);
        fp = print;
```

```
        (*fp)(10);
        fp(20);
        return 0;
}
void print(int value)
{
        printf("\n %d", value);
}
```

PASSING A FUNCTION POINTER AS AN ARGUMENT TO A FUNCTION

A function pointer can be passed as a function's calling argument. This is done when you want to pass a pointer to a callback function.

Example:

```
include <stdio.h>
int add(int, int);
int subt(int, int);
int operate(int (*operate_fp) (int, int), int, int);
main()
{       int result;
        result = operate(add, 9, 7);
        printf ("\n Addition Result = %d", result);
        result = operate(sub, 9, 7);
        printf ("\n Subtraction Result = %d", result);
}
int add (int a, int b)
{
 return (a+b);
}
int subtract (int a, int b)
{
  return (a-b);
}
```

```
int operate(int (*operate_fp) (int, int), int a, int b)
{
        int result;
        result = (*operate_fp) (a,b);
        return result;
}
```

FUNCTIONS WITH VARIABLE NUMBER OF ARGUMENTS

Sometimes, we may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... )
{
   . .
}
int main()
{
  func(1, 2, 3);
  func(1, 2, 3, 4);
}
```

It should be noted that function **func()** has last argument as ellipses i.e. three dotes (**...**) and the one just before the ellipses is always an **int** which will represent total number variable arguments passed. To use such functionality you need to make use of **stdarg.h** header file which provides functions and macros to implement the functionality of variable arguments and follow the following steps:

- Define a function with last parameter as ellipses and the one just before the ellipses is always an **int** which will represent number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Now let us follow the above steps and write down a simple function which can take variable number of parameters and returns their average:

```
#include <stdio.h>
#include <stdarg.h>
double average(int num,...)
{
   va_list valist;
   double sum = 0.0;
   int i;
   /* initialize valist for num number of arguments */
   va_start(valist, num);
   /* access all the arguments assigned to valist */
   for (i = 0; i < num; i++)
   {
      sum += va_arg(valist, int);
   }
   /* clean memory reserved for valist */
   va_end(valist);
   return sum/num;
}
int main()
{
  printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
  printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

When the above code is compiled and executed, it produces the following result. It should be noted that the function **average()** has been called twice and each time first argument represents the total number of variable arguments being passed. Only ellipses will be used to pass variable number of arguments.

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

UNIT-II

STRUCTURE

A structure is same as that of records. It stores related information about an entity.

Structure is basically a user defined data type that can store related information (even of different data types) together.

A structure is declared using the keyword struct followed by a structure name. All the variables of the structures are declared within the structure. A structure type is defined by using the given syntax.

```
struct struct-name
{       data_type var-name;
        data_type var-name;
};
struct student
{       int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

The structure definition does not allocates any memory. It just gives a template that conveys to the C compiler how the structure is laid out in memory and gives details of the member names. Memory is allocated for the structure when we declare a variable of the structure. For ex, we can define a variable of student by writing,

```
struct student stud1;
```

TYPEDEF DECLARATIONS

When we precede a struct name with typedef keyword, then the struct becomes a new type. It is used to make the construct shorter with more meaningful names for types already defined by C or for types that you have declared. With a typedef declaration, becomes a synonym for the type.

For example, writing

```
typedef struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

Now that you have preceded the structure's name with the keyword typedef, the student becomes a new data type. Therefore, now you can straight away declare variables of this new data type as you declare variables of type int, float, char, double, etc. to declare a variable of structure student you will just write,

```
student stud1;
```

INITIALIZATION OF STRUCTURES

Initializing a structure means assigning some constants to the members of the structure.

When the user does not explicitly initializes the structure then C automatically does that. For int and float members, the values are initialized to zero and char and string members are initialized to the '\0' by default.

The initializers are enclosed in braces and are separated by commas. Note that initializers match their corresponding types in the structure definition.

The general syntax to initialize a structure variable is given as follows.

```
struct struct_name
{       data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .....
}struct_var = {constant1, constant2, constant 3,...};
```

OR

```
struct struct_name
```

```
{       data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .....
};
```

```
struct struct_name struct_var = {constant1, constant2, ….};
struct student stud1 = {01, “Rahul”, “BCA”, 45000};
```

ACCESSING THE MEMBERS OF A STRUCTURE

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a ‘.’ (dot operator).

The syntax of accessing a structure a member of a structure is:

struct_var.member_name

For ex, to assign value to the individual data members of the structure variable Rahul, we may write,

```
stud1.r_no = 01;
strcpy(stud1.name, “Rahul”);
stud1.course = “BCA”;
stud1.fees = 45000;
```

We can assign a structure to another structure of the same type. For ex, if we have two structure variables stu1 and stud2 of type struct student given as

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
struct student stud2;
```

Then to assign one structure variable to another we will write,

```
stud2 = stud1;
```

Write a program using structures to read and display the information about a student

```
#include<stdio.h>
int main()
{       struct student
        {       int roll_no;
                char name[80];
                float fees;
                char DOB[80];
        };
        struct student stud1;
        printf("\n Enter the roll number : ");
        scanf("%d", &stud1.roll_no);
        printf("\n Enter the name : ");
        scanf("%s", stud1.name);
        printf("\n Enter the fees : ");
        scanf("%f", &stud1.fees);
        printf("\n Enter the DOB : ");
        scanf("%s", stud1.DOB);
        printf("\n ROLL No. = %d", stud1.roll_no);
        printf("\n NAME. = %s", stud1.name);
        printf("\n FEES. = %f", stud1.fees);
        printf("\nDOB. = %s", stud1.DOB);
}
```

Output:

```
ROLL No. 12
NAME. YYY
FEES. 20000
13.1.1990
```

Program using structure to read and display the information about an employee.

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

struct details
{
 char name[30];
 int age;
 char address[500];
 float salary;
};

int main()
{
 struct details detail;
 clrscr();
 printf("\nEnter name:\n");
 gets(detail.name);
 printf("\nEnter age:\n");
 scanf("%d",&detail.age);
 printf("\nEnter Address:\n");
 gets(detail.address);
 printf("\nEnter Salary:\n");
 scanf("%f",&detail.salary);
printf("Name of the Employee : %s \n",detail.name);
 printf("Age of the Employee : %d \n",detail.age);
 printf("Address of the Employee : %s \n",detail.address);
 printf("Salary of the Employee : %f \n",detail.salary);
 getch();
}
```

OUTPUT:

Name of the Employee: XXX
Age of the Employee : 26
Address of the Employee : CHENNAI
Salary of the Employee:12000

1. NESTED STRUCTURES

A structure can be placed within another structure. That is, a structure may contain another structure as its member. Such a structure that contains another structure as its member is called a nested structure.

Write a program to read and display information of a student using structure within a structure

```
#include<stdio.h>

int main()

{       struct DOB

        {

                int day;

                int month;

                int year;

        };

        struct student

        {       int roll_no;

                char name[100];

                float fees;

                struct DOB date;

        };

        struct student stud1;

        printf("\n Enter the roll number : ");

        scanf("%d", &stud1.roll_no);

        printf("\n Enter the name : ");

        scanf("%s", stud1.name);

        printf("\n Enter the fees : ");

        scanf("%f", &stud1.fees);

        printf("\n Enter the DOB : ");
```

```
        scanf("%d %d %d", &stud1.date.day, &stud1.date.month, &stud1.date.year);
        printf("\n ********STUDENT'S DETAILS *******");
        printf("\n ROLL No. = %d", stud1.roll_no);
        printf("\n NAME. = %s", stud1.name);
        printf("\n FEES. = %f", stud1.fees);
        printf("\n DOB = %d - %d - %d", stud1.date.day, stud1.date.month,
    stud1.date.year);
    }
```

2. ARRAYS OF STRUCTURES

The general syntax for declaring an array of structure can be given as,

struct struct_name struct_var[index];

Example:

struct student stud[30];

Now, to assign values to the i^{th} student of the class, we will write,

```
stud[i].r_no = 09;
stud[i].name = "RASHI";
stud[i].course = "MCA";
stud[i].fees = 60000;
```

Write a program to read and display information of all the students in the class.(Array of Structures

```
#include<stdio.h>
int main()
{       struct student
        {
                int roll_no;
                char name[80];
```

```
        float fees;
        char DOB[80];
    };
    struct student stud[50];
    int n, i;
    printf("\n Enter the number of students : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {   printf("\n Enter the roll number : ");
        scanf("%d", &stud[i].roll_no);
        printf("\n Enter the name : ");
        scanf("%s", stud[i].name);
        printf("\n Enter the fees : ");
        scanf("%f", stud[i].fees);
        printf("\n Enter the DOB : ");
        scanf("%s", stud[i].DOB);
    }
    for(i=0;i<n;i++)
    {   printf("\n ********DETAILS OF %dth STUDENT*******", i+1);
        printf("\n ROLL No. = %d", stud[i].roll_no);
        printf("\n NAME. = %s", stud[i].name);
        printf("\n ROLL No. = %f", stud[i].fees);
        printf("\n ROLL No. = %s", stud[i].DOB);
    }
}
```

3. STRUCTURES AND FUNCTIONS

Passing Individual Structure Members to a Function

To pass any individual member of the structure to a function we must use the direct selection operator to refer to the individual members for the actual parameters. The called program does not know if the two variables are ordinary variables or structure members.

```
#include<stdio.h>
struct student
{
        int rno;
        char name[10];
}stud1;
void display(int, int);
main()
{
        student stud1 = {2, “Ram”};
        display(stud1.rno,stud1.name);
        return 0;
}
void display(int rno, char name)
{
        printf("%d %s", rno,name);
}
```

Passing A Structure To A Function

When a structure is passed as an argument, it is passed using **call by value** method. That is a copy of each member of the structure is made. No doubt, this is a very inefficient method especially when the structure is very big or the function is called frequently. Therefore, in such a situation passing and working with pointers may be more efficient.

The general syntax for passing a structure to a function and returning a structure can be given as, struct struct_name func_name(struct struct_name struct_var);

The code given below passes a structure to the function using **call-by-value** method.

```
#include<stdio.h>
struct student
{
        int rno;
        char name[20];
}stud1;
void display(struct student);
main()
{
        student stud1 = {2, “Ram”};
        display(stud1);
        return 0;
}
void display( struct student stud1)
{
        printf("%d %c", stud1.rno, stud1.name);
}
```

Passing Structures Through Pointers

C allows to create a pointer to a structure. Like in other cases, a pointer to a structure is never itself a structure, but merely a variable that holds the address of a structure. The syntax to declare a pointer to a structure can be given as

```
struct struct_name
{
        data_type member_name1;
```

```
        data_type member_name2;
        .....
};*ptr;
```

OR

```
struct struct_name *ptr;
```

For our student structure we can declare a pointer variable by writing

```
struct student *ptr_stud, stud;
```

The next step is to assign the address of stud to the pointer using the address operator(&). So to assign the address, we will write

```
ptr_stud = &stud;
```

To access the members of the structure, one way is to write

```
/* get the structure, then select a member */
(*ptr_stud).roll_no;
```

An alternative to the above statement can be used by using 'pointing-to' operator (->) as shown below.

```
/* the roll_no in the structure ptr_stud points to */
ptr_stud->roll_no = 01;
```

Write a program using pointer to structure to initialize the members in the structure.

```
#include<stdio.h>
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
```

```
};
main()
{       struct student stud1,  *ptr_stud1;
        ptr_stud1 = &stud1;
        ptr_stud1->r_no = 01;
        strcpy(ptr_stud1->name, "Rahul");
        strcpy(ptr_stud1->course, "BCA");
        ptr_stud1->fees = 45000;
        printf("\n DETAILS OF STUDENT");
        printf("\n ----------------------------------------------");
        printf("\n ROLL NUMBER = %d", ptr_stud1->r_no);
        printf("\n NAME  = ", puts(ptr_stud1->name));
        printf("\n COURSE  = ", puts(ptr_stud1->course));
        printf("\n FEES = %f", ptr_stud1->fees);
}
```

4. SELF REFERENTIAL STRUCTURES

Self referential structures are those structures that contain a reference to data of its same type. That is, a self referential structure in addition to other data contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.

```
struct node
{
        int val;
        struct node *link;
};
```

Here the structure node will contain two types of data- an integer val and next that is a pointer to a node. You must be wondering why do we need such a structure? Actually, self-referential structure is the foundation of other data structures.

Eg:

```
struct node
{
int data;
struct node *link;
}
void main()
{
struct item *n1;
scanf("%d",&n1->data);
n1->link=NULL;
printf("%d%s",n1->data,n1->link);
}
```

UNION

Like structure, a union is a collection of variables of different data types. The only difference between a structure and a union is that in case of unions, you can only store information in one field at any one time.

To better understand union, think of it as a chunk of memory that is used to store variables of different types. When a new value is assigned to a field, the existing data is replaced with the new data.

Thus unions are used to save memory. They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time.

DECLARING A UNION

The syntax for declaring a union is same as that of declaring a structure.

```
union union-name
{       data_type var-name;
        data_type var-name;
        ...
};
```

Again, the typedef keyword can be used to simplify the declaration of union variables.

The most important thing to remember about a union is that the size of an union is the size of its largest field. This is because a sufficient number of bytes must be reserved to store the largest sized field.

ACCESSING A MEMBER OF A UNION

A member of a union can be accessed using the same syntax as that of a structure. To access the fields of a union, use the dot operator(.). That is the union variable name followed by the dot operator followed by the member name.

INITIALIZING UNIONS

- It is an error to initialize any other union member except the first member
- A striking difference between a structure and a union is that in case of a union, the field fields share the same memory space, so fresh data replaces any existing data. Look at the code given below and observe the difference between a structure and union when their fields are to be initialized.

```
#include<stdio.h>

typedef struct POINT1

{       int x, y;

};

typedef union POINT2

{

        int x;

        int y;

};

main()

{       POINT1 P1 = {2,3};

        // POINT2 P2 ={4,5}; Illegeal with union


        POINT2 P2;

        P2. x = 4;

        P2.y = 5;
```

```
        printf("\n The co-ordinates of P1 are %d and %d", P1.x, P1.y);
        printf("\n The co-ordinates of P2 are %d and %d", P2.x, P2.y);
        return 0;
}
```

OUTPUT

The co-ordinates of P1 are 2 and 3

The co-ordinates of P2 are 5 and 5

ARRAYS OF UNION VARIABLES

Like structures we can also have array of union variables. However, because of the problem of new data overwriting existing data in the other fields, the program may not display the accurate results.

```
#include <stdio.h>
union POINT
{
int x, y;
 };
main()
{
int i;
union POINT points[3];
points[0].x = 2;
points[0].y = 3;
points[1].x = 4;
points[1].y = 5;
points[2].x = 6;
points[2].y = 7;
```

```
for(i=0;i<3;i++)
{
 printf("\n Co-ordinates of Points[%d] are %d and %d", i, points[i].x, points[i].y);
}
return 0;
}
```

OUTPUT

- Co-ordinates of Points[0] are 3 and 3
- Co-ordinates of Points[1] are 5 and 5
- Co-ordinates of Points[2] are 7 and 7

UNIONS INSIDE STRUCTURES

Union can be very useful when declared inside a structure. Consider an example in which you want a field of a structure to contain a string or an integer, depending on what the user specifies. The following code illustrates such a scenario.

```
struct student
{   union
    {       char name[20];
            int roll_no;
    };
    int marks;
};
main()
{
    struct student stud;
    char choice;
    printf("\n You can enter the name or roll number of the student");
    printf("\n Do you want to enter the name? (Yes or No) : ");
```

```
    gets(choice);
    if(choice=='y' || choice=='Y')
    {       printf("\n Enter the name : ");
        gets(stud.name);
    }
    else
    {       printf("\n Enter the roll number : ");
            scanf("%d", &stud.roll_no);
    }
    printf("\n Enter the marks : ");
    scanf("%d", &stud.marks);
    if(choice=='y' || choice=='Y')
            printf("\n Name : %s ", stud.name);
    else
            printf("\n Roll Number : %d ", stud.roll_no);
    printf("\n Marks : %d", stud.marks);
}
```

ENUMERATED DATA TYPES

The enumerated data type is a user defined type based on the standard integer type.

- An enumeration consists of a set of named integer constants. That is, in an enumerated type, each integer value is assigned an identifier. This identifier (also known as an enumeration constant) can be used as symbolic names to make the program more readable.
- To define enumerated data types, enum keyword is used.
- Enumerations create new data types to contain values that are not limited to the values fundamental data types may take. The syntax of creating an enumerated data type can be given as below.

  enum enumeration_name { identifier1, identifier2, ….., identifiern };

Consider the example given below which creates a new type of variable called COLORS to store colors constants.

- enum COLORS {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE}; In case you do not assign any value to a constant, the default value for the first one in the list - RED (in our case), has the value of 0. The rest of the undefined constants have a value 1 more than its previous one. So in our example,
- RED = 0, BLUE = 1, BLACK = 2, GREEN = 3, YELLOW = 4, PURPLE = 5, WHITE =6

If you want to explicitly assign values to these integer constants then you should specifically mention those values as shown below.

- enum COLORS {RED = 2, BLUE, BLACK = 5, GREEN = 7, YELLOW, PURPLE , WHITE = 15};

As a result of the above statement, now

- RED = 2, BLUE = 3, BLACK = 5, GREEN = 7, YELLOW = 8, PURPLE = 9, WHITE = 15

COMPARING ENUMERATED TYPES

- C also allows using comparison operators on enumerated data type
- Since enumerated types are derived from integer type, they can be used in a switch-case statement.

```
enum {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE}bg_color;

switch(bg_color)

{

    case RED:

    case BLUE:

    case GREEN:

                printf("\n It is a primary color");

                break;

    case default:

                printf("\n It is not a primary color");

                break;

}
```

Command line arguments in C

main() function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,

argc - Number of arguments in the command line including program name
argv[] –is a pointer array which points to each argument passed to the program

Example 1:

```
#include <stdio.h>

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

{

   int x;

   printf("%d\n",argc);

   for (x=0; x<argc; x++)

      printf("%s\n",argv[x]);

   return 0;

}
```

Example 2:

```
#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");
  }
}
```

FILE HANDLING

A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. There are two kinds of files.

• Text file
• Binary file

STREAMS IN C

- In C, the standard streams are termed as pre-connected input and output channels between a text terminal and the program (when it begins execution). Therefore, stream is a logical interface to the devices that are connected to the computer.
- Stream is widely used as a logical interface to a file where a file can refer to a disk file, the computer screen, keyboard, etc.
- The three standard streams in C languages are- standard input (stdin), standard output (stdout) and standard error (stderr).

  Standard input (stdin): Standard input is the stream from which the program receives its data. The program requests transfer of data using the *read* operation. However, not all programs require input. Generally, unless redirected, input for a program is expected from the keyboard.

  Standard output (stdout): Standard output is the stream where a program writes its output data. The program requests data transfer using the *write* operation. However, not all programs generate output.

  Standard error (stderr): Standard error is basically an output stream used by programs to report error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. No doubt, the *standard output* and *standard error* can also be directed to the same destination.

BUFFER ASSOCIATED WITH FILE STREAM

- When a stream linked to a disk file is created, a buffer is automatically created and associated with the stream. A buffer is nothing but a block of memory that is used for temporary storage of data that has to be read from or written to a file.
- Buffers are needed because disk drives are block oriented devices as they can operate efficiently when data has to be read/ written in blocks of certain size. The size of ideal buffer size is hardware dependant.
- The buffer acts as an interface between the stream (which is character-oriented) and the disk hardware (which is block oriented). When the program has to write data to the stream, it is saved in the buffer till it is full. Then the entire contents of the buffer are written to the disk as a block.

USING FILES IN C

- To use files in C, we must follow the steps given below.
- Declare a file pointer variable
- Open the file
- Process the file
- Close the file

TYPES OF FILES

- In C, the types of files used can be broadly classified into two categories- text files and binary files**.**

1. TEXT FILE

Text files can store character data. A text file can be thought of as a stream of characters that can be processed sequentially. A text file is usually opened for reading, writing and appending operation. They have a .txt extension.

Mode	**Meaning**
r	Opens a text file for reading
w	Create a text file for writing
a	Append to a text file
r +	Opens a text file for read \ write
w +	Creates a text file for read \ write
a +	Append or create a text file for read \ write

Declaring a file pointer variable

- There can be a number of files on the disk. In order to access a particular file, you must specify the name of the file that has to be used. This is accomplished by using a file pointer variable that points to a structure FILE (defined in stdio.h). The file pointer will then be used in all subsequent operations in the file. The syntax for declaring a file pointer is

  FILE *file_pointer_name;

  For example, if we write

  FILE *fp;

  Then, fp is declared as a file pointer.

- An error will be generated if you use the filename to access a file rather than the file pointer

Opening a File

- A file must be first opened before data can be read from it or written to it. In order to open a file and associate it with a stream, the fopen() function is used. The prototype of fopen() can be given as:
- FILE *fopen(const char *file_name, const char *mode);
- Using the above prototype, the file whose pathname is the string pointed to by file_name is opened in the mode specified using the mode. If successful, fopen() returns a pointer-to-structure and if it fails, it returns NULL.

WRITING DATA TO FILES

C provides the following set of functions to writing data to a file.

Functions used for writing data to a **Text** file

fprintf()

fputw()

fputs()

fputc()

Functions used for writing data to a **Binary** file

fwrite()

READ DATA FROM FILES

C provides the following set of functions to read data from a file.

Functions used for reading data from a **Text** file

fscanf()

fgetw()

fgets()

fgetc()

Functions used for reading data from a **Binary** file

fread()

Files read and write:(Formatted input / output file function)

Once the file has been successfully opened, it can be read using file functions.
Formatted input / output file function:

fscanf()

The fscanf() is used to read formatted data from the stream. The **syntax** of the fscanf() can be given as,

Syntax:

int fscanf(FILE *stream, const char *format,…);

The fscanf() is used to read data from the *stream* and store them according to the parameter *format* into the locations pointed by the additional arguments.

Eg:
fscanf (fp1,"control string", &id.name, &salary);
Here the values for id, name &salary variable are read from the input file pointed by fp1.

fprintf()

- The fpritnt() is used to write formatted output to stream. Its syntax can be given as,

Syntax:

int fprintf (FILE * stream, const char * format, ...);

- The function writes to the specified *stream*, data that is formatted as specified by the *format* argument. After the *format* parameter, the function can have as many additional arguments as specified in *format*.

Eg:
fprintf (fp2,"%d %s %f", id, name, salary);

To illustrate formatted input/output file functions

```
#include <stdio.h>
#include <conio.h>
void main ()
{
int rollno;
char name [15];
FILE * fp1, *fp2;
fp1=fopen ("input.txt", "r");
fp2=fopen ("output.txt","w");
while (! feof (fp1))
```

```
{
fscanf (fp1, "%d %s", &rollno, name);
fprintf (fp2, "%d %s %n", rollno, name);
}
fclose (fp1);
fclose (fp2);
}
```

Output:

Input.txt
1001 arun
1002 babu
1003 chitra
Output.txt
1001 arun
1002 babu
1003 chitra

Files read and write:(UnFormatted input / output file function)

a) The fgetc () function:

The fgetc () function reads a single character from the file stream. The fgetc() function returns the next character from stream, or EOF if the end of file is reached or if there is an error. The syntax of fgetc() can be given as

int fgetc(FILE *stream);

Eg:

ch = fgetc (fp1)

Here fp1 is the file pointer returned by fopen () function. Ch is the character variable fgetc () function reads a single character from the file pointer by fp1 and assigns it to the character variable ch.

b) fputc () function:

The fputc() is used to write a character to the stream(File pointer).

int fputc(int *c*, FILE **stream*);

Eg:

fputc (ch, fp2);
fputc () function write the value of ch into the file pointed by fp2.

c) getw () function

Syntax:

var = getw (fptr);
Eg:
Num = getw (fp1);

The getw () function reads an integer value from the file printed by fp1 and assigns the value to the integer variable Num.

d) The putw () function:
The putw () function writes an integer value into the file pointed by the file pointer fptr.

Syntax:
putw (Num, fptr);
Eg:
putw (Num, fp2);

Here putw () function writes the integer value in the variables Num into the file pointed by fp2.

e) fgets () function:
fgets () function used to read a string from a file pointed by fptr. The function fgets () takes three arguments. the first arguments is the address where the read. String is to be stored, the second is the maximum length of the string and the third argument is the file pointer.

Syntax:

char *fgets(char *str, int *size*, FILE **stream*);

Eg:

fgets (str1, 20, fp1);

Here fp1 is the file pointer str1 is the array variable. This function reads a string of 20 characters from the file associated with the file pointer fp1 and store in str1. This returns a string of characters.

f) fputs () function:
It writes the string to the file pointed by fptr.

Syntax:

int fputs(const char *str, FILE *stream);

Eg:
fputs (str2, fp2);

File close:
Once the read and write operations in a file are completed, the file is closed.

Syntax:
fclose (fptr);
Eg:
fclose (fp1);

Example Program To illustrate unformatted input / output functions:

(or)

Example program for Reading and Writing data in a Text File

```
#include <stdio.h>
#include <conio.h>
void main ()
{
int regno;
char name [15];
char grade;
FILE * fp1, *fp2;
fp1 = fopen (“input.txt”, “r”);
fp2 = fopen (“output.txt”, “w”);
While (! Feof (fp1))
{
regno = getw (fp1);
grade = fgetc (fp1);
fgets (name, 10, fp1);
putw (reg _ no, fp2);
fputc (grade, fp2);
fputs (name, fp2);
}
fclose (fp1);
fclose (fp2);
}
```

Input.txt:
1001 arun S
1002 babu A
1003 chitra S

Output.txt:
1001 arun S
1002 babu A
1003 chitra S

2. BINARY FILES

A binary file is a collection of data stored in the internal format of the computer. This data can be an integer, a floating point number, a character, an array or any other structured data. Unlike text files, binary files contain data that are meaningful only if they are properly interpreted by a program. If the data are textual, 1 byte represents one character. But if the data are numeric, 2 or more bytes are considered as a data item.

Opening and closing of binary file:
Like a text file, a binary file is also opened using fopen function.

Mode	**Meaning**
rb	Opens a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r + b	Opens a binary file for read \ write
w + b	Creates a binary file for read \ write
a + b	Append or create a binary file for read \ write

Binary file operations:

a) The fread () function:
The fread() function is used to read data from a file. It is an unformatted read function.
Its syntax can be given as

Syntax:

int fread(void *str, size, num, FILE *stream);

The function fread() reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by str. The data is read from the given input stream.

Eg:
Fread (& buffer _ record, n, 2, fp1);

Here fread () function reads n bytes of 2 blocks from the file pointed by fp2 and store in a memory region called buffer _ record.

b) The fwrite () function:

- The *fwrite()* is used to write data to a file. The syntax of fwrite can be given as,

  Syntax:

  int fwrite(const void *str, size, count, FILE *stream);

- The *fwrite*() function will write, from the array pointed to by s*tr*, up to count objects of size specified by *size*, to the stream pointed to by *stream*.
- The file-position indicator for the stream (if defined) will be advanced by the number of bytes successfully written. In case of error, the error indicator for the stream will be set.

Eg:
fwrite (& buffer _ rec, n, 2, fp2);

Here fwrite () function writes n bytes of 2 blocks from the buffer _ rec to the file pointed by fp2.

Example Program To illustrate binary file input\output function(Reading and Writing data in a Binary file)

```
#include <stdio.h>
#include <conio.h>
struct student
{
int rollno;
char name [15];
};
void main ()
{
struct student s1[10], s2[10];
FILE * fp1, *fp2;
int i, n=5;
clrscr ();
fp1 = fopen (“binary.txt”, wb);
if (fp1! = NULL)
{
for (i = 0; i <n;i++)
{
fscanf (fp2, “%d %s”, & s1 [i].rollno , s1[i].name)
}
for (i=0; i<n; i++)
{
fwrite (& s1, sizeof (s1), 1, fp1);
}
}
fclose (fp1);
fp1 = fopen (“binary.txt”, “rb”);
```

```
if (fp1! = NULL)
{
fread (&s2, sizeof (s2), 1, fp1);
for (i=0; i<n; i++)
{
printf ("%d \t %s \n", s2 [i].rollno, s2 [i].name);
}
}
fclose (fp1);
}
```

RANDOM ACCESS TO FILES

FUNCTIONS FOR SELECTING A RECORD RANDOMLY

fseek()

- fseek() is used to reposition a binary stream. The prototype of fseek() can be given as,

  Syntax:

  int fseek(FILE *stream, long offset, int origin);

- fseek() is used to set the file position pointer for the given stream. Offset is an integer value that gives the number of bytes to move forward or backward in the file. Offset may be positive or negative, provided it makes sense. For example, you cannot specify a negative offset if you are starting at the beginning of the file. The *origin* value should have one of the following values (defined in stdio.h):

  SEEK_SET (0): to perform input or output on offset bytes from start of the file

  SEEK_CUR (1): to perform input or output on offset bytes from the current position in the file

  SEEK_END (2): to perform input or output on offset bytes from the end of the file

Note: SEEK_SET, SEEK_CUR and SEEk_END are defined constants with value 0, 1 and 2 respectively.

- On successful operation, fseek() returns zero and in case of failure, it returns a non-zero value. For example, if you try to perform a seek operation on a file that is not opened in binary mode then a non-zero value will be returned.

- fseek() can be used to move the file pointer beyond a file, but not before the beginning.

Write a program to print the records in reverse order. The file must be opened in binary mode. Use fseek()

```
#include<stdio.h>
#include<conio.h>
main()
{   typedef struct employee
    {       int emp_code;
            char name[20];
            int hra;
            int da;
            int ta;
    };
    FILE *fp;
    struct employee e;
    int result, i;
    fp =  fopen("employee.txt", "rb");
    if(fp==NULL)
    {       printf("\n Error opening file");
            exit(1);
    }
    for(i=5;i>=0;i--)
    {       fseek(fp, i*sizeof(e), SEEK_SET);
            fread(&e, sizeof(e), 1, fp);
            printf("\n  EMPLOYEE CODE : %d", e.emp_code);
            printf("\n Name : %s", e.name);
            printf("\n HRA, TA and DA : %d %d %d", e.hra, e.ta, e.da);
    }
```

```
    fclose(fp);
    getch();
    return 0;
}
```

rewind()

- rewind() is used to adjust the position of file pointer so that the next I/O operation will take place at the beginning of the file. It's Syntax can be given as

  Syntax:

  void rewind(FILE *f);

- rewind() is equivalent to calling fseek() with following parameters: fseek(f,0L,SEEK_SET);

fgetpos()

- The fgetpos() is used to determine the current position of the stream. It's Syntax can be given as

  Syntax:

  int fgetpos(FILE *stream, fpos_t *pos);

- Here, stream is the file whose current file pointer position has to be determined. pos is used to point to the location where fgetpos() can store the position information. The pos variable is of type fops_t which is defined in stdio.h and is basically an object that can hold every possible position in a FILE.
- On success, fgetpos() returns zero and in case of error a non-zero value is returned. Note that the value of pos obtained through fgetpos() can be used by the fsetpos() to return to this same position.

fsetpos()

- The fsetpos() is used to move the file position indicator of a stream to the location indicated by the information obtained in "pos" by making a call to the fgetpos(). Its Syntax is

  Syntax:

  int fsetpos(FILE *stream, const fops_t pos);

- Here, stream points to the file whose file pointer indicator has to be re-positioned. pos points to positioning information as returned by "fgetpos".
- On success, fsetpos() returns a zero and clears the end-of-file indicator. In case of failure it returns a non-zero value

remove()

- The remove() as the name suggests is used to erase a file. The Syntax of remove() as given in stdio.h can be given as,

  Syntax:

  int remove(const char *filename);

- The remove() will erase the file specified by filename. On success, the function will return zero and in case of error, it will return a non-zero value.

Renaming the File

- The rename() as the name suggests is used to renames a file. The Syntax is:

  Syntax:

  int rename(const char *oldname, const char *newname)

- Here, the *oldname* specifies the pathname of the file to be renamed and the *newname* gives the new pathname of the file.
- On success, rename() returns zero. In case of error, it will return a non-zero value will set the *errno* to indicate the error.

ftell()

- The ftell function is used to know the current position of file pointer. It is at this position at which the next I/O will be performed. The syntax of the ftell() defined in stdio.h can be given as:

  Syntax:

  long ftell (FILE *stream);

- On successful, ftell() function returns the current file *position* (in bytes) for stream. However, in case of error, ftell() returns -1.
- When using ftell(), error can occur either because of two reasons:
- First, using ftell() with a device that cannot store data (for example, keyboard)
- Second, when the position is larger than that can be represented in a long integer. This will usually happen when dealing with very large files

Example Program to illustrate Random Access files -Use fseek (), ftell (), rewind ()

```
#include <stdio.h>
#include <conio.h>
void main ()
{
FILE *fp1;
clrscr ();
fp1 = open (“inputfile.txt”, “r”);
if (fp1! = NULL)
{
printf (“\n the file pointer is at the location: %d”, ftell (fp1));
fseek (fp1, 13, 0);
\\ the file pointer moves 13+1 bytes forward from the beginning of the file
printf (“\n the file pointer is at the location: %d”, ftell (fp1));
fseek (fp1, 4, 1);
\\ the file pointer moves 4+1 bytes forward from the current position of the file pointer
printf (“\n the file pointer is at the location: %d”, ftell (fp1));
fseek (fp1, 0, 2);
\\ the file pointer moves to the end of the file
printf (“\n the file pointer is at the location: %d”, ftell (fp1));
fseek (fp1, -10, 2);
\\ the file pointer moves 10+1 bytes backward from the end of file
printf (“\n the file pointer is at the location: %d”, ftell (fp1));
rewind (fp1);
\\ resets the file pointer to the beginning of the file
printf (“\n the file pointer is at the location: %d”, ftell (fp1));
}
}
```

Inputfile.txt:

Random Access File

Output:

The file pointer is at the location 0
The file pointer is at the location 13
The file pointer is at the location 17
The file pointer is at the location 18
The file pointer is at the location 8
The file pointer is at the location 0

DETECTING THE END-OF-FILE

- In C, there are two ways to detect the end-of-file
- While reading the file in text mode, character by character, the programmer can compare the character that has been read with the EOF, which is a symbolic constant defined in stdio.h with a value -1.

```
while(1)
{   c = fgetc(fp);  // here c is an int variable
    if (c==EOF)
            break;
    printf("%c", c);
 }
```

- The other way is to use the standard library function feof() which is defined in stdio.h. The feof() is used to distinguish between two cases
    - When a stream operation has reached the end of a file
    - When the EOF ("end of file") error code has been returned as a generic error indicator even when the end of the file has not been reached

Syntax:

int feof(FILE *fp);

- feof() returns zero (false) when the end of file has not been reached and a one (true) if the end-of-file has been reached.

Program to illustrate feof () function:

```
#include <stdio.h>
void main ()
{
FILE * fp1;
fp1 = fopen (“Inputfile.txt”, “r”);
while (! feof (fp1))
{
fscanf (fp1, “%d %s”, &rikk _ no, name);
printf (“The roll no is %d”, roll _ no);
printf (“Name is %s”, name);
}
fclose (fp1);
}
```

FILE MANIPULATION

1. C program to create a file & store information.

```
#include <stdio.h>
void main()
{
   FILE *fp;
   char name[20];
   int age;
   float salary;

   /*  open for writing */
   fp = fopen("emp.rec", "w");
   if (fp == NULL)
   {
      printf("File does not exists \n");
      return;
   }
   printf("Enter the name \n");
   scanf("%s", name);
   fprintf(fp, "Name= %s\n", name);
   printf("Enter the age\n");
   scanf("%d", &age);
   fprintf(fp, "Age= %d\n", age);
   printf("Enter the salary\n");
   scanf("%f", &salary);
   fprintf(fp, "Salary  = %.2f\n", salary);
   fclose(fp);
}
```

Output:

```
Enter the name
raj
Enter the age
40
Enter the salary
4000000
```

2. C Program illustrates reading of data from a file

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   FILE *fp;
   char filename[15];
   char ch;
   printf("Enter the filename to be opened \n");
   scanf("%s", filename);
   /*  open the file for reading */
   fp = fopen(filename, "r");
   if (fp == NULL)
   {
      printf("Cannot open file \n");
      exit(0);
   }
   ch = fgetc(fp);
   while (ch != EOF)
   {
      printf ("%c", ch);
      ch = fgetc(fp);
   }
   fclose(fp);
}
```

Output:

```
Enter the filename to be opened
pgm.c
```

3. C Program delete a specific line from a text file.

```
#include <stdio.h>
 int main()
{
   FILE *fileptr1, *fileptr2;
   char filename[40];
   char ch;
   int delete_line, temp = 1;

   printf("Enter file name: ");
   scanf("%s", filename);
   //open file in read mode
```

```
  fileptr1 = fopen(filename, "r");
  ch = getc(fileptr1);
` while (ch != EOF)
  {
     printf("%c", ch);
     ch = getc(fileptr1);
  }
  //rewind
  rewind(fileptr1);
  printf(" \n Enter line number of the line to be deleted:");
  scanf("%d", &delete_line);
  //open new file in write mode
  fileptr2 = fopen("replica.c", "w");
  ch = getc(fileptr1);
  while (ch != EOF)
  {
     ch = getc(fileptr1);
     if (ch == '\n')
        temp++;
        //except the line to be deleted
        if (temp != delete_line)
        {
           //copy all lines in file replica.c
           putc(ch, fileptr2);
        }
  }
  fclose(fileptr1);
  fclose(fileptr2);
  remove(filename);
  //rename the file replica.c to original name
  rename("replica.c", filename);
  printf("\n The contents of file after being modified are as follows:\n");
  fileptr1 = fopen(filename, "r");
  ch = getc(fileptr1);
  while (ch != EOF)
  {
     printf("%c", ch);
     ch = getc(fileptr1);
  }
  fclose(fileptr1);
  return 0;
}
```

Output:

```
Enter file name: pgm1.c
```

4. C Program appends the content of file at the end of another.(Merging of Two Files into another file)

This C program merges two files and stores their contents in another file. The files which are to be merged are opened in read mode and the file which contained content of both the files is opened in write mode. To merge two files, first we open a file and read it character by character and close the read content in another file, then we read the contents of another file and store it in file, we read two files until EOF is reached.

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   FILE *fp1, *fp2, *ftemp;
   char ch, file1[20], file2[20], file3[20];
   printf("Enter name of first file ");
   gets(file1);
   printf("Enter name of second file ");
   gets(file2);
   printf("Enter name to store merged file ");
   gets(file3);
   fp1 = fopen(file1, "r");
   fp2 = fopen(file2, "r");
   if (fsring1 == NULL || fsring2 == NULL)
   {
      perror("Error has occured");
   }
   ftemp = fopen(file3, "w");
   if (ftemp == NULL)
   {
      perror("Error has occures");
   }
   while ((ch = fgetc(fp1)) != EOF)
      fputc(ch, ftemp);
   while ((ch = fgetc(fp2) ) != EOF)
      fputc(ch, ftemp);
   printf("Two files merged  %s successfully.\n", file3);
   fclose(fp1);
   fclose(fp2);
   fclose(ftemp);
   return 0;
}
```

Output:

Enter name of first **file** a.txt
Enter name of second **file** b.txt
Enter name to store merged **file** merge.txt
Two files merged merge.txt successfully.

5. C Program displays the number of lines in a text file.

```
#include <stdio.h>
 int main()
{
   FILE *fileptr;
   int count_lines = 0;
   char filechar[40], chr;
    printf("Enter file name: ");
   scanf("%s", filechar);
   fileptr = fopen(filechar, "r");
  //extract character from file and store in chr
   chr = getc(fileptr);
   while (chr != EOF)
   {
      //Count whenever new line is encountered
      if (chr == 'n')
      {
         count_lines = count_lines + 1;
      }
      //take next character from file.
      chr = getc(fileptr);
   }
   fclose(fileptr); //close file.
   printf("There are %d lines in %s  in a file\n", count_lines, filechar);
   return 0;
}
```

Output:

Enter **file** name: pgm2.c
There are 43 lines **in** pgm2.c **in** a **file**

6. Program to copy content of one file to another using command line arguments

```
void main(int argc, char *argv[])
{
FILE * fp1,*fp2;
char buffer [100];
if (argc!=3){
printf("Usage: copyfile inFile outFile");
return 0;
}
fp1 = fopen (argv[1], "r");
fp2 = fopen (argv[2], "w");
if (fp1== NULL)
perror ("Error opening input file");
else
{
while ( ! feof (fp1) )
{
fgets (buffer , 100 , fp1);
fputs (buffer , fp2);
}
fclose (fp1);
fclose (fp2);
}
}
```

7.C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.

```
#include <stdio.h>
int main()
{
  char name[50];
  int marks,i,n;
  printf("Enter number of students: ");
  scanf("%d",&n);
  FILE *fptr;
  fptr=(fopen("student.txt","a"));
  if(fptr==NULL
{
     printf("Error!");
     exit(1);
  }
```

```
  for(i=0;i<n;++i)
  {
    printf("For student%d\nEnter name: ",i+1);
    scanf("%s",name);
    printf("Enter marks: ");
    scanf("%d",&marks);
    fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
  }
  fclose(fptr);
  return 0;
}
```

8.C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.

```
#include <stdio.h>
struct s
{
char name[50];
int height;
};
int main(){
   struct s a[5],b[5];
   FILE *fptr;
   int i;
   fptr=fopen("file.txt","wb");
   for(i=0;i<5;++i)
   {
      fflush(stdin);
      printf("Enter name: ");
      gets(a[i].name);
      printf("Enter height: ");
      scanf("%d",&a[i].height);
   }
   fwrite(a,sizeof(a),1,fptr);
   fclose(fptr);
   fptr=fopen("file.txt","rb");
   fread(b,sizeof(b),1,fptr);
   for(i=0;i<5;++i)
   {
      printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
   }
   fclose(fptr);
}
```

THE <STDIO.H> HEADER FILE

1. **FUNCTIONS :**

clearerr

Resets error indication.

clrscr

Clears the screen and resets the print position.

fclose

Closes a stream.

feof

Tests a stream for an end-of-file indicator.

ferror

Tests a stream for a read or write error.

fflush

Flushes a stream.

fgetc

Function version of fgetc.

fgetchar

Function version of getchar.

fgetpos

Gets the current file pointer position.

fgets

Gets a string from a stream.

fopen

Opens a stream.

fprintf

Sends formatted output to a stream.

fputc

Function version of fputc.

fputchar

Function version of putchar.

fputs

Outputs a string to a stream.

fread

Reads data from a stream.

freopen

Associates a new file with an open stream.

fscanf

File parsing function.

fseek

Repositions the file pointer of a stream.

fsetbufsize

Sets the buffer size of a file.

fsetpos

Positions the file pointer of a stream.

ftell

Returns the current file pointer.

fwrite

Writes data to a stream.

getc

Gets a character from a stream.

getchar

Gets a character from the keyboard (with echoing to the screen).

gets

Gets a string from the keyboard.

getsn

Gets a string from the keyboard avoiding buffer overflows.

printf_xy

Sends formatted output to the fixed place on the screen.

printf

Sends formatted output to the screen.

putc

Writes a character to a stream.

putchar

Outputs a character to the screen in TTY mode.

puts

Outputs a string to the screen in TTY mode.

remove

Macro that removes a file.

rename

Renames a file.

rewind

Repositions file pointer to stream's beginning.

scanf

Console input parsing function.

unlink

Deletes a file.

2. CONSTANTS

EOF

Indicates that the end of a file has been reached.

NULL

A null-pointer value.

TMP_MAX

Contains the maximum number of temporary file names.

3. PREDEFINED TYPES

FILE

A structure describing an opened file.

FileFlags

An enumeration describing internal flags for file handling.

fpos_t

A type describing the current position in a file.

SeekModes

An enumeration for describing possible modes used in fseek.

size_t

A type to define sizes of strings and memory blocks.

va_list

A void pointer which can be interpreted as an argument list.

EXPLANATION FOR ALL FUNCTIONS

clearerr

```
void clearerr (FILE *stream);
```

Resets error indication.

clearerr resets the error and end-of-file indicators of the stream associated to the structure pointed to by *stream* to 0. Once the error indicator is set, stream operations continue to return error status until a call is made to clearerr or rewind.

---

clrscr

```
void clrscr (void);
```

Clears the screen and resets the print position.

Function clrscr is very similar to stantard TIOS function ClrScr (defined in graph.h header file). The difference is in fact that clrscr moves the current print/plot position to (0, 0), but with ClrScr the current print/plot position remains intact. More precise, clrscr calls ClrScr then MoveTo passing two zeros as arguments. Always use clrscr instead of ClrScr if you want to use TTY printing functions like puts, printf etc.

---

fclose

```
short fclose (FILE *stream);
```

Short is the data type.(short int)

Closes a stream.

fclose closes the stream associated to the structure pointed to by *stream*. In this implementation, fclose unlocks the file (which is locked during it is opened), and frees the file descriptor structure pointed to by *stream*. fclose returns 0 on success. It returns EOF if any errors were detected.

---

feof

```
short feof (FILE *stream);
```

Tests a stream for an end-of-file indicator.

feof is a macro that tests the stream associated to the structure pointed to by *stream* for an end-of-file indicator. Once the indicator is set, read operations on the file return the indicator until fseek, fsetpos or rewind is called, or until the stream is closed. feof returns nonzero if an end-of-file indicator was detected on the last (usually input) operation on the given stream. It returns 0 if end-of-file has not been reached.

---

ferror

```
short ferror (FILE *stream);
```

Tests a stream for a read or write error.

ferror is a macro that tests the stream associated to the structure pointed to by *stream* for a read or write error. It the stream's error indicator has been set, it remains set (and all file I/O operations will return error) until clearerr or rewind is called, or until the stream is closed. ferror returns nonzero if an error was detected on the named stream.

---

fflush

```
short fflush (FILE *stream);
```

Flushes a stream.

fflush is supposed to flush the output buffer for *stream* to the associated file if the given stream has buffered output. As the TI file system is not buffered (of course), fflush has no effect, except for undoing the effect of the ungetc function. fflush returns 0 on success (which is always the case on the TI).

---

fgetc

```
short fgetc (FILE *stream);
```

Function version of fgetc.

fgetc is usually equal to getc, except fgetc is implemented as a function, but getc is implemented as a macro.

---

fgetchar

```
short fgetchar (void);
```

Function version of getchar.

fgetchar is equal to getchar, except fgetchar is implemented as a function, but getchar is implemented as a macro.

---

fgetpos

```
short fgetpos (FILE *stream, fpos_t *pos);
```

Gets the current file pointer position.

fgetpos stores the position of the file pointer associated with the stream associated with the structure pointed to by *stream* in the location pointed to by *pos* The exact value is irrelevant. On success, fgetpos returns 0. On failure, it returns a nonzero value.

Note: fgetpos is implemented here as a macro which calls ftell.

---

fgets

```
char *fgets (char *s, short n, FILE *stream);
```

Gets a string from a stream.

fgets reads characters from stream associated to the structure pointed to by *stream* into the string *s*. It does this by calling fgetc repeatedly. The function stops reading when it reads either *n* - 1 characters or a '\r' (0x0D) character, whichever comes first. fgets retains the newline character at the end of *s*, eventually translated to '\n' character if the stream is opened in "text" mode (see fopen). A null byte is appended to *s* to mark the end of the string. On success, fgets returns the string pointed to by *s*. It returns NULL in a case of error.

Note: fgets is used mainly with files opened in "text" mode. As an example, this command may be useful for reading a text line from a TEXT variable.

---

fopen

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

Opens a stream.

fopen opens the file named by *filename* (this is a normal C string, which should be in lowercase) and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations. The *mode* string used in calls to fopen is one of the following values:

Mode	Description
r	Open for reading only.
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end of file, or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open for update at the end of the file, or create if the file does not exist.

On successful completion, fopen returns a pointer to the newly opened stream. In the event of error, it returns NULL. Note that files on TI are in fact TIOS variables, so their maximal size is limited (as the size of the variable is limited). Maybe in the future I will try to implement files which are limited only by the amount of the free memory.
To specify that a given file is being opened or created in text mode, append a 't' to the mode string ("rt", "w+t", and so on). Similarly, to specify binary mode, append a 'b' to the mode string ("wb", "a+b", and so on). What "text" or "binary" exactly means will be explained a bit later. fopen also allows the t or b to be inserted between the letter and the '+' character in the

mode string. For example, "rt+" is equivalent to "r+t". If a 't' or 'b' is not given in the mode string, 't' is assumed (this slightly differs from the ANSI convention: in ANSI C the mode in this case is governed by the global variable **_fmode**, which is not implemented here).

fprintf

```
short fprintf (FILE *stream, const char *format, ...);
```

Sends formatted output to a stream.

fprintf sends formatted output to a string. In fact, it does the following:

- accepts a series of arguments;
- applies to each argument a format specifier contained in the format string pointed to by *format* (see printf for details on format specifiers);
- outputs the formatted data to the stream associated with the structure pointed to by *stream*

There must be the same number of format specifiers as arguments. fprintf returns the number of bytes output. In the event of error, it returns EOF.

---

fputc

```
short fputc (short c, FILE *stream);
```

Function version of fputc.

fputc is usually equal to putc, except fputc is implemented as a function, but putc is implemented as a macro.

---

fputchar

```
short fputchar (short c);
```

Function version of putchar.

fputchar is usually equal to putchar, except fputchar is implemented as a function, but putchar is implemented as a macro.

---

fputs

```
short fputs (const char *s, FILE *stream);
```

Outputs a string to a stream.

fputs copies the null-terminated string *s* to the output stream associated to the structure pointed to by *stream*. It does this by calling putc repeatedly. It does not append a newline character, and the terminating null character is not copied. On successful completion, fputs returns the last character written. Otherwise, it returns a value of EOF.

---

fread

```
unsigned short fread (void *ptr, unsigned short size, unsigned short n, FILE *stream);
```

Reads data from a stream.

fread reads *n* items of data, each of length *size* bytes, from the input stream associated with the structure pointed to by *stream* into a block pointed to by *ptr*. The total number of bytes read is *n* x *size*. fread fread returns the number of items (not bytes) actually read. If the operation was sucessful, the returned result should be equal to *n*. In a case of error, returned result will be smaller (possibly zero).

Note: fread is proposed to be used in "binary" mode (see fopen). Although this is not strictly necessary, it is is highly recommended opening *stream* in "binary" mode if you want to use this function. Anyway, there will not be any character translations during reading, even if the file is opened in "text" mode.

---

freopen

```
FILE *freopen (const char *filename, const char *mode, FILE *stream);
```

Associates a new file with an open stream.

freopen substitutes the named file in place of the open stream. It closes stream, regardless of whether the open succeeds. In this implementation, freopen is implemented as macro which first calls fclose passing *stream* to it, then calls fopen passing *filename* and *mode* to it. Such implementation is not absolutely correct, because the address of the file descriptor structure may be changed after closing and reopening again (if a garbage collect occurs). This is not a problem in programs which uses freopen as in

f=freopen (name, mode, f);

but it might cause problems in programs which uses freopen as in

freopen (name, mode, f);

To solve this problem, freopen macro will always re-assign the variable *f* to a (eventually) new value, so both above examples will be correct (the only small problem is in fact that *f* must ultimately be an lvalue, i.e, a variable or something similar).

On successful completion, freopen returns the argument *stream* (possibly changed). In the event of error, it returns NULL.

---

fscanf

```
short fscanf(FILE *file, const char *format, ...);
```

File parsing function.

reads the input from the file of type char *.

---

fseek

```
short fseek (FILE *stream, long offset, short whence);
```

Repositions the file pointer of a stream.

fseek sets the file pointer associated with *stream* to a new position that is *offset* bytes from the file location given by *whence*. For text mode streams (see fopen), *offset* should be 0 or a value returned by ftell. *whence* must be one of the following values (defined in enum SeekModes):

whence	File location
SEEK_SET	File beginning
SEEK_CUR	Current file pointer position
SEEK_END	End-of-file

fseek discards any character pushed back using ungetc. fseek returns 0 if the pointer is successfully moved. It returns a nonzero value on failure.

---

fsetbufsize

```
void fsetbufsize (unsigned short newsize, FILE *f);
```

Sets the buffer size of a file.

fsetbufsize sets the buffer size of an open file. The buffer size determines how much memory is reallocated to the file every time a write needs more memory from the heap. The default size (128 bytes) is set when the file is opened and should be sufficient for most uses. Setting a larger value will make writes faster at the cost of possibly running out of memory prematurely. If *newsize* is zero or f is NULL, no changes will be made.

---

fsetpos

```
short fsetpos (FILE *stream, const fpos_t *pos);
```

Positions the file pointer of a stream.

fsetpos sets the file pointer associated with *stream* to a new position (given in the variable pointed to by *pos*). The new position is the value obtained by a previous call to fgetpos on that stream. It also clears the end-of-file indicator on the file that stream points to and undoes any effects of ungetc on that file. On success, fsetpos returns 0. On failure, it returns a nonzero value.

---

ftell

```
long ftell (const FILE *stream);
```

Returns the current file pointer.

ftell returns the current file pointer for the stream associated with the structure pointed to by *stream*. The offset is measured in bytes from the beginning of the file. The value returned by ftell can be used in a subsequent call to fseek. ftell returns the current file pointer position on success. It returns EOF on error.

---

fwrite

```
unsigned short fwrite (const void *ptr, unsigned short size, unsigned short n, FILE *stream);
```

Writes data to a stream.

fwrite writes *n* items of data, each of length *size* bytes, to the output file associated with the structure pointed to by *stream*. The data written begins at *ptr*. The total number of bytes written is *n* x *size*. *ptr* in the declarations is a pointer to any object. fwrite returns the number of items (not bytes) actually written. If the operation was sucessful, the returned result should

be equal to *n*. In a case of error, returned result will be smaller (possibly zero).

Note: fwrite is proposed to be used in "binary" mode (see fopen). Although this is not strictly necessary, it is is highly recommended opening *stream* in "binary" mode if you want to use this function. Anyway, there will not be any character translations during writing, even if the file is opened in "text" mode.

getc

```
#define getc fgetc
```

Gets a character from a stream.

getc gets the next character on the given input stream (associated with the structure pointed to by *stream*), and increments the stream's file pointer to point to the next character. If the file is opened in "text" mode (see fopen), a character after '\r' will be swallowed during reading (to skip over the "command byte" at the begining of the each line in a TEXT variable).

On success, getc returns the character read, after converting it to an integer without sign extension. On error (usually end-of-file), it returns EOF.

---

getchar

```
#define getchar fgetchar
```

Gets a character from the keyboard (with echoing to the screen).

fgetchar returns the character read from the keyboard. It is similar to ngetchx except getchar shows a cursor (a simple underscore), echoes the character read on the screen and supports the CHAR menu. '\r' character (i.e. ENTER key) will be echoed as a "new line".

gets

```
char *gets (char *string);
```

Gets a string from the keyboard.

gets collects a string of characters terminated by a new line from the keyboard (by repeated calling to getchar) and puts it into *string*. The new line is replaced by a null character ('\0') in *string*. gets returns when it encounters a new line (i.e. when the ENTER key is pressed); everything up to the new line is copied into *string*. gets returns the string argument *string* (ANSI proposes returning of NULL in a case of error, but this never occurs on the TI). For editing, the backspace key is supported.

getsn

```
char *getsn (char *string, unsigned long maxlen);
```

Gets a string from the keyboard avoiding buffer overflows.

getsn works like gets, but with a maximum length specified. The maximum length (maxlen) is the total size of the buffer. In other words, the null-terminator is included in the maximum length. When the buffer is full (i.e. when the string length is maxlen - 1), getsn will not accept any more characters. Only backspace or ENTER are allowed in that situation. Here is an example of usage:

```
char buffer[50];
int a, b;
clrscr ();
puts ("A = ");
a = atoi (getsn (buffer, 50));
puts ("B = ");
b = atoi (getsn (buffer, 50));
printf ("%d + %d = %d", a, b, a+b);
```

atoi is an ANSI C standard function from stdlib.h header file.

Note: getsn is not an ANSI C standard function, but the equivalent of fgets (buffer, maxlen, stdin) in ANSI C. It is needed because terminal streams are not implemented in TIGCCLIB.

printf_xy

```
void printf_xy (short x, short y, const char *format, ...);
```

Sends formatted output to the fixed place on the screen.

printf_xy is similar to the standard ANSI C printf function, except:

- this function displays formatted output to the screen at the strictly specified position, more precise, starting from the point (*x*, *y*);
- text printed with printf_xy will not wrap at the right end of the screen (if the text is longer, the result is unpredictable);
- characters '\n' will not be translated to "new line";
- this function will never cause screen scrolling;
- current print/plot position remains intact after executing this function.

printf_xy is a GNU C macro which calls sprintf and DrawStr.

---

printf

```
void printf (const char *format, ...);
```

Sends formatted output to the screen.

printf is nearly full implementation of standard ANSI C printf function, which sends the formatted output to the screen in terminal (TTY) mode. In fact, it does the following:

- accepts a series of arguments;
- applies to each a format specifier contained in the format string pointed to by *format*;
- outputs the formatted data to the screen.

putc

```
#define putc fputc
```

Writes a character to a stream.

putc writes the character c to the stream given by *stream*. It will update the stream's file pointer, and expands the size of associated variable if necessary. If the file is opened in "text" mode (see fopen), all '\n' characters will be translated to '\r' 0x20 sequence during writting (to satisfy the format of the text in TEXT variables). On success, putc returns the character *c*. On error, it returns EOF.

---

putchar

```
#define putchar fputchar
```

Outputs a character to the screen in TTY mode.

Outputs a character *c* to the screen in TTY mode. This means the following:

- The current print position will be moved in the text line after printing the last character in the screen line;
- after printing the last character in the last screen line, the screen will scroll upwards;
- characters '\n' will be translated to "next line" (and this is the only control code which has a special implementation);
- the current print position will be updated after the character is printed.

---

puts

```
void puts (const char *s);
```

Outputs a string to the screen in TTY mode.

puts outputs the null-terminated string *s* to the screen by repeated calling to putchar until the end of the string is reached.

---

remove

```
#define remove unlink
```

Macro that removes a file.

remove deletes the file specified by *filename*. It is a macro that simply translates its call to a call to unlink (name known from UNIX). So, both remove and unlink are equal. Although ANSI C proposes rename, unlink is more common in UNIX programs.

rename

```
short rename (const char *oldname, const char *newname);
```

Renames a file.

rename changes the name of a file from *oldname* to *newname* (both filenames are normal C strings, which should be in lowercase). Filenames may also contain folder names. Folder names in *oldname* and *newname* need not be the same, so rename can be used to move a file from one folder to another. On successfully renaming the file, rename returns 0. In the event of error, EOF is returned.

Note: Function SymMove from vat.h header file is very similar like rename, except the parameters and returned result are somewhat different. As rename is not a TIOS entry and SymMove is, the usage of SymMove is recommended instead of rename (although SymMove is not ANSI standard).

---

rewind

```
void rewind (FILE *stream);
```

Repositions file pointer to stream's beginning.

rewind (*stream*) is equivalent to

fseek (*stream*, 0, SEEK_SET)

except that rewind clears the end-of-file and error indicators, while fseek only clears the end-of-file indicator.

scanf

```
short scanf (const char *format, ...);
```

Console input parsing function.

Works like sscanf, but reads the input from the keyboard using getsn rather than from a buffer of type char *. [ENTER] is interpreted as EOF. The amount of possible user input is limited by available memory. scanf may also return 0 if there was no memory left at all to allocate the temporary buffer.

tmpnam

```
char *tmpnam (char *s);
```

Produces a unique random file name.

tmpnam returns a random file name of 8 characters which does not exist on the calculator. If *s* is NULL, tmpnam returns a pointer to a static buffer, otherwise it fills *s* and returns a pointer to it. When passing NULL to tmpnam, it is best to treat the pointer returned as if it were pointing to constant data. It is assumed that the buffer pointed to by *s* is at least 9 bytes long.

tmpnam is capable of returning TMP_MAX or 25^8 combinations. When nearing TMP_MAX, performance decreases significantly, and eventually, the function will run into an infinite loop. These factors, however, should not pose any problems for the currently supported calculator platforms. You will run into the maximum number of handles a lot sooner.

ungetc

```
short ungetc (short c, FILE *stream);
```

Pushes a character back into input stream.

ungetc pushes the character *c* back onto the stream associated with the structure pointed to by *stream*. This character will be returned on the next call to getc (or related functions like fread) for that stream. A second call to ungetc without a call to getc will force the previous character to be forgotten. A call to fflush, fseek, fsetpos, or rewind erases all memory of any pushed-back characters. ungetc returns the character pushed back. ANSI C proposes that it need to return EOF if the operation fails, but in this implementation it cannot fail.

unlink

```
short unlink (const char *filename);
```

Deletes a file.

unlink deletes the file specified by *filename* (it is a normal C string, which should be in lowercase). If your file is open, be sure to close it before removing it. The string pointed to by *filename* may include a folder name too. On successful completion, unlink returns 0. On error, it returns EOF.

EOF

```
#define EOF (-1)
```

Indicates that the end of a file has been reached.

EOF is a constant which is usually returned as the result of file handling functions if an end-of-file is reached, or in a case of an error. The ANSI standard does not propose exact value of this constant, but it proposes that it must be negative.

FILE

```
typedef struct
{
  …..
} FILE;
```

A structure describing an opened file.

FILE is the main file control structure for streams. The exact structure of it is very platform-dependent, so ANSI C proposes that the exact structure of this structured type should not be known, and well-written programs do not need to access the internal fields of this structure.

fpos_t

```
typedef unsigned long fpos_t;
```

A type describing the current position in a file.

fpos_t is a scalar type used for saving the current file position using fgetpos and restoring it back using fsetpos.

```
SeekModesenum SeekModes {SEEK_SET, SEEK_CUR, SEEK_END};
```

An enumeration for describing possible modes used in fseek.