Problem Solving Techniques Using C

UNIT – I

1.0 Introduction

A computer is a very powerful and versatile machine capable of performing a multitude of different tasks, yet it has no intelligence or thinking power. The intelligence Quotient (I.Q) of a computer is zero. A computer performs many tasks exactly in the same manner as it is told to do. This places responsibility on the user to instruct the computer in a correct and precise manner, so that the machine is able to perform the required job in a proper way. A wrong or ambiguous instruction may sometimes prove disastrous.

In order to instruct a computer correctly, the user must have clear understanding of the problem to be solved. A part from this he should be able to develop a method, in the form of series of sequential steps, to solve it. Once the problem is well-defined and a method of solving it is developed, then instructing he computer to solve the problem becomes relatively easier task.

Thus, before attempt to write a computer program to solve a given problem. It is necessary to formulate or define the problem in a precise manner. Once the problem is defined, the steps required to solve it, must be stated clearly in the required order.

1.1 Procedure (Steps Involved in Problem Solving)

A computer cannot solve a problem on its own. One has to provide step by step solutions of the problem to the computer**.** In fact, the task of problem solving is not that of the computer. It is the programmer who has to write down the solution to the problem in terms of simple operations which the computer can understand and execute.

In order to solve a problem by the computer, one has to pass though certain stages or steps. They are

1. Understanding the problem
2. Analyzing the problem
3. Developing the solution
4. Coding and implementation.

1. Understanding the problem: Here we try to understand the problemto be solved in totally. Before with the next stage or step, we should be absolutely sure about the objectives of the given problem.

2. Analyzing the problem: After understanding thoroughly the problemto be solved, we look different ways of solving the problem and evaluate each

of these methods. The idea here is to search an appropriate solution to the problem under consideration. The end result of this stage is a broad overview of the sequence of operations that are to be carries out to solve the given problem.

3. **Developing the solution:** Here the overview of the sequence ofoperations that was the result of analysis stage is expanded to form a detailed step by step solution to the problem under consideration.

4. **Coding and implementation:** The last stage of the problem solving isthe conversion of the detailed sequence of operations in to a language that the computer can understand. Here each step is converted to its equivalent instruction or instructions in the computer language that has been chosen for the implantation.

1.2 Algorithm

Definition

A set of sequential steps usually written in Ordinary Language to solve a given problem is called **Algorithm**.

It may be possible to solve to problem in more than one ways, resulting in more than one algorithm. The choice of various algorithms depends on the factors like reliability, accuracy and easy to modify. The most important factor in the choice of algorithm is the time requirement to execute it, after writing code in High-level language with the help of a computer. The algorithm which will need the least time when executed is considered the best.

Steps involved in algorithm development

An algorithm can be defined as **"a complete, unambiguous, finite numberof logical steps for solving a specific problem "**

Step1. Identification of input: For an algorithm, there are quantities tobe supplied called input and these are fed externally. The input is to be indentified first for any specified problem.

Step2: Identification of output: From an algorithm, at least one quantityis produced, called for any specified problem.

Step3 : Identification the processing operations : All the calculationsto be performed in order to lead to output from the input are to be identified in an orderly manner.

Step4 : Processing Definiteness : The instructions composing thealgorithm must be clear and there should not be any ambiguity in them.

Step5 : Processing Finiteness : If we go through the algorithm, then forall cases, the algorithm should terminate after a finite number of steps.

Step6 : Possessing Effectiveness : The instructions in the algorithmmust be sufficiently basic and in practice they can be carries out easily.

Example

1. Suppose we want to find the average of three numbers, the algorithm is as follows

Step 1 Read the numbers a, b, c

Step 2 Compute the sum of a, b and c

Step 3 Divide the sum by 3

Step 4 Store the result in variable d

Step 5 Print the value of d

Step 6 End of the program

1.2.2 Algorithms for Simple Problem

Write an algorithm for the following

1. Write an algorithm to calculate the simple interest using the

   formula. Simple interest = P*N* R/100.

   Where P is principle Amount, N is the number of years and R is the rate of interest.

   Step 1: Read the three input quantities' P, N and R.

   Step 2 : Calculate simple interest as

   Simple interest = P* N* R/100

   Step 3: Print simple interest.

   Step 4: Stop.

FLOW CHART

A flow chart is a step by step diagrammatic representation of the logic paths to solve a given problem**.** Or A flowchart is visual or graphical representation of an algorithm.

The flowcharts are pictorial representation of the methods to b used to solve a given problem and help a great deal to analyze the problem and

plan its solution in a systematic and orderly manner. A flowchart when translated in to a proper computer language, results in a complete program.

Advantages of Flowcharts

1. The flowchart shows the logic of a problem displayed in pictorial fashion which felicitates easier checking of an algorithm.

2. The Flowchart is good means of communication to other users. It is also a compact means of recording an algorithm solution to a problem.

3. The flowchart allows the problem solver to break the problem into parts. These parts can be connected to make master chart.

4. The flowchart is a permanent record of the solution which can be consulted at a later time.

1.4 Symbols used in Flow-Charts

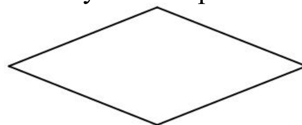
The symbols that we make use while drawing flowcharts as given below are as per conventions followed by International Standard Organization (ISO).

a. Oval: Rectangle with rounded sides is used to indicate either START/STOP of the program.

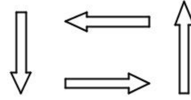
b. Input and output indicators: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are represented in these Parallelograms.

c. Process Indicators: - Rectangle is used to indicate any set of processing operation such as for storing arithmetic operations.

d. Decision Makers: The diamond is used for indicating the step of decision making and therefore known as decision box. Decision boxes are used to test the conditions or ask questions and depending upon the answers, the appropriate actions are taken by the computer. The decision box symbol is

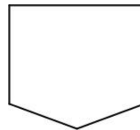


e. Flow Lines: Flow lines indicate the direction being followed in theflowchart. In a Flowchart, every line must have an arrow on it to indicate the direction. The arrows may be in any direction



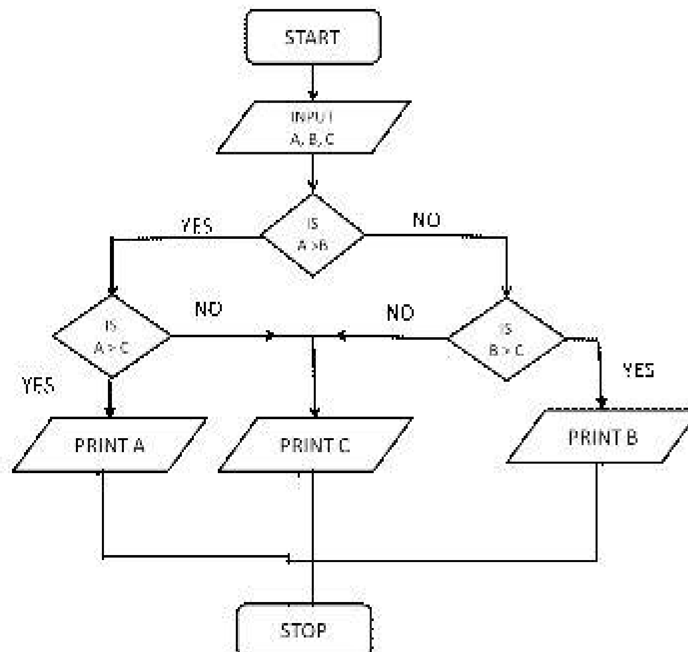
f. On- Page connectors: Circles are used to join the different parts of aflowchart and these circles are called on-page connectors. The uses of these connectors give a neat shape to the flowcharts. Ina complicated problems, a flowchart may run in to several pages. The parts of the flowchart on different pages are to be joined with each other. The parts to be joined are indicated by the circle.

g. Off-page connectors: This connector represents a break in the path of flowchart which is too large to fit on a single page. It is similar to on-page connector. The connector symbol marks where the algorithm ends on the first page and where it continues on the second.



1.4.1 Simple Problems using Flow Chart

1. Draw a flowchart to find out the biggest of the three unequal positive numbers.



2.0 Introduction

'C' is high level language and is the upgraded version of another language (Basic Combined Program Language). C language was designed at Bell laboratories in the early 1970's by Dennis Ritchie. C being popular in the modern computer world can be used in Mathematical Scientific, Engineering and Commercial applications

The most popular Operating system UNIX is written in C language. This language also has the features of low level languages and hence called as "System Programming Language"

Features of C language

- Simple, versatile, general purpose language
- It has rich set of Operators
- Program execution are fast and efficient
- Can easily manipulates with bits, bytes and addresses
- Varieties of data types are available
- Separate compilation of functions is possible and such functions can be called by any C program
- Block- structured language
- Can be applied in System programming areas like operating systems, compilers & Interpreters, Assembles, Text Editors, Print Spoolers, Network Drivers, Modern Programs, Data Bases, Language Interpreters, Utilities etc.

2.1 Character Set

The character set is the fundamental raw-material for any language. Like natural languages, computer languages will also have well defined character-set, which is useful to build the programs.

The C language consists of two character sets namely – source character set execution character set. Source character set is useful to construct the statements in the source program. Execution character set is employed at the time of execution of h program.

1. **Source character set :**This type of character set includes three typesof characters namely alphabets, Decimals and special symbols.

i. Alphabets : A to Z, a to z and Underscore(_)

ii. Decimal digits : 0 to 9

iii. Special symbols: + - * / ^ % = & ! ( ) { } [ ] " etc

2. Execution character set : This set of characters are also called asnon-graphic characters because these are invisible and cannot be printed or displayed directly.

These characters will have effect only when the program being executed. These characters are represented by a back slash (\) followed by a character.

Execution character	Meaning	Result at the time of execution
\ n	End of a line	Transfers the active position of cursor to the initial position of next line
\ 0 (zero)	End of string	Null
\ t	Horizontal Tab	Transfers the active position of cursor to the next Horizontal Tab
\ v	Vertical Tab	Transfers the active position of cursor to the next Vertical Tab
\ f	Form feed	Transfers the active position of cursor to the next logical page
\ r	Carriage return	Transfers the active position of cursor to the initial position of current line

2.2 Structure of a 'C' Program

The Complete structure of C program is

The basic components of a C program are:

- main()
- pair of braces { }
- declarations and statements
- user defined functions

Preprocessor Statements*:*These statements begin with # symbol. Theyare called preprocessor directives. These statements direct the C preprocessor to include header files and also symbolic constants in to C program. Some of the preprocessor statements are

#include<stdio.h>: for the standard input/output functions

#include<test.h>: for file inclusion of header file Test.

#define NULL 0: for defining symbolic constant NULL = 0 etc.

Global Declarations*:*Variables or functions whose existence is known inthe main() function and other user defined functions are called global variables (or functions) and their declarations is called global declaration. This declaration should be made before main().

main(): As the name itself indicates it is the main function of every C program.Execution of C program starts from main (). No C program is executed without main() function. It should be written in lowercase letters and should not be terminated by a semicolon. It calls other Library functions user defined functions. There must be one and only one main() function in every C program.

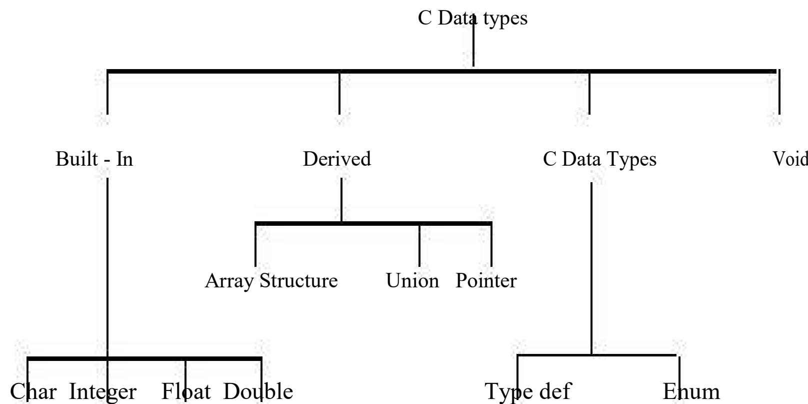
Braces*:*Every C program uses a pair of curly braces ({,}0. The leftbrace indicates beginning of main() function. On the other hand, the right brace indicates end of the main() function. The braces can also be used to indicate the beginning and end of user-defined functions and compound statements.

Declarations*:*It is part of C program where all the variables, arrays,functions etc., used in the C program are declared and may be initialized with their basic data types.

Statements*:*These are instructions to the specific operations. They maybe input-output statements, arithmetic statements, control statements and other statements. They are also including comments.

User-defined functions*:*These are subprograms. Generally, a subprogramis a function, and they contain a set of statements to perform a specific task. These are written by the user; hence the name is user-defined functions. They may be written before or after the main() function.

2.3 Data Types in 'C'



The built-in data types and their extensions is the subject of this chapter. Derived data types such as arrays, structures, union and pointers and user defined data types such as typedef and enum.

Basic Data Types

There are four basic data types in C language. They are Integer data, character data, floating point data and double data types.

a. Character data: Any character of the ASCII character set can beconsidered as a character data types and its maximum size can be 1 byte or 8 byte long. 'Char' is the keyword used to represent character data type in C.

Char - a single byte size, capable of holding one character.

b. Integer data: The keyword 'int' stands for the integer data type in Cand its size is either 16 or 32 bits. The integer data type can again be classified as

1. Long int - long integer with more digits
2. Short int - short integer with fewer digits.
3. Unsigned int - Unsigned integer
4. Unsigned short int – Unsigned short integer
5. Unsigned long int – Unsigned long integer

As above, the qualifiers like short, long, signed or unsigned can be applied to basic data types to derive new data types.

int - an Integer with the natural size of the host machine.

c. Floating point data: - The numbers which are stored in floating pointrepresentation with mantissa and exponent are called floating point (real) numbers. These numbers can be declared as 'float' in C.

float – Single – precision floating point number value.

d. Double data : - Double is a keyword in C to represent double precisionfloating point numbers.

double - Double – precision floating point number value.

Data Kinds in C

Various data kinds that can be included in any C program can fall in to the following.

a. Constants/Literals
b. Reserve Words Keywords
c. Delimeters

d. Variables/Identifiers

a. Constans/Literals: Constants are those, which do not change, duringthe execution of the program. Constants may be categorized in to:

- Numeric Constants
- Character Constants
- String Constants

1. Numeric Constants

Numeric constants, as the name itself indicates, are those which consist of numerals, an optional sign and an optional period. They are further divided into two types:

(a) Integer Constants (b) Real

Constants

a. Integer Constants

A whole number is an integer constant Integer constants do not have a decimal point. These are further divided into three types depending on the number systems they belong to. They are:

i. Decimal integer constants
ii. Octal integer constants
iii. Hexadecimal integer constants

i. A decimal integer constant is characterized by the following properties

- It is a sequence of one or more digits ([0…9], the symbols of decimal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid decimal integer constants:

456

-123

Some examples of invalid decimal integer constants:

4.56 - Decimal point is not permissible

1,23 - Commas are not permitted

ii. An octal integer constant is characterized by the following properties

- It is a sequence of one or more digits ([0…7], symbols of octal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the digit 0.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid octal integer constants:

0456

-0123

+0123

Some examples of invalid octal integer constants:

04.56 - Decimal point is not permissible

04,56 - Commas are not permitted

x34 - x is not permissible symbol

568 - 8 is not a permissible symbol

iii. An hexadecimal integer constant is characterized by the following properties

- It is a sequence of one or more symbols ([0…9][A….Z], the symbols of Hexadecimal number system).
- It may have an optional + or - sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the symbols 0X or 0x.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid hexadecimal integer constants:

0x456

-0x123

0x56A

Some examples of invalid hexadecimal integer constants:

0x4.56 - Decimal point is not permissible

0x4,56 - Commas are not permitted.

b. Real Constants

The real constants also known as *floating point constants* are written in two forms:

i. Fractional Form

The real constants in Fractional form are characterized by the following characteristics:

- Must have at least one digit.
- Must have a decimal point.
- May be positive or negative and in the absence of sign taken as positive.
- Must not contain blanks or commas in between digits.
- May be represented in exponential form, if the value is too higher or too low.

Some examples of valid real constants:

456.78

-123.56

Some examples of invalid real constants:

4.56 - Blank spaces are not permitted

4,56 - Commas are not permitted

456 - Decimal point missing

ii. Exponential Form

The exponential form offers a convenient way for writing very large and small real constant. For example, 56000000.00, which can be written as 0.56 *, 10^{8} is written as *0.56E8* or *0.56e8* in exponential form. 0.000000234, which can be written as 0.234 * 10^{-6} is written as *0.234E-6* or *0.234e-6* in exponential form. The letter E or e stand for exponential form.

A real constant expressed in exponential form has two parts: (i) Mantissa part, (ii) Exponent part. Mantissa is the part of the real constant to the left of *E* or *e*, and the Exponent of a real constant is to the right of *E*or *e*. Mantissa and Exponent of the above two number are shown below.

The real constants in exponential form and characterized by the following characteristics:

- The mantissa must have at least one digit.
- The mantissa is followed by the letter E or e and the exponent.
- The exponent must have at least one digit and must be an integer.
- A sign for the exponent is optional.

Some examples of valid real constants:

3E4

23e-6

0.34E6

Some examples of invalid real constants:

23E - No digit specified for exponent

23e4.5 - Exponent should not be a fraction

23,4e5 - Commas are not allowed

256*e8- * not allowed

2. Character Constants

Any character enclosed with in single quotes (‘) is called character constant.

A character constant:

- May be a single alphabet, single digit or single special character placed with in single quotes.
- Has a maximum length of 1 character.

- ‘C’
- ‘c’
- ‘:’
- ‘*’

3. String Constants

A string constant is a sequence of alphanumeric characters enclosed in double quotes whose maximum length is 255 characters.

Following are the examples of valid string constants:

- “My name is Krishna”
- “Bible”
- “Salary is 18000.00”

Following are the examples of invalid string constants:

My name is Krishna - Character are not enclosed in double quotation marks.

“My name is Krishna - Closing double quotation mark is missing.

‘My name is Krishna’ - Characters are not enclosed in double quotation marks

b. Reserve Words/Keywords

In C language , some words are reserved to do specific tasks intended for them and are called Keywords or Reserve words. The list reserve words are

auto	do	goto
break	double	if
case	else	int
char	extern	long
continue	float	register
default	for	return
short	sezeof	static
struct	switch	typedef
union	unsigned	void
while	const	entry
violate	enum	noalias

c. Delimiters

This is symbol that has syntactic meaning and has got significance. These will not specify any operation to result in a value. C language delimiters list is given below

Symbol	Name	Meaning
#	Hash	Pre-processor directive
,	comma	Variable delimiter to separate variable
:	colon	label delimiter
;	Semicolon	statement delimiter
( )	parenthesis	used for expressions
{ }	curly braces	used for blocking of statements
[ ]	square braces	used along with arrays

d. Variables / Identifiers

These are the names of the objects, whose values can be changed during the program execution. Variables are named with description that transmits the value it holds.

[A quantity of an item, which can be change its value during the executionof program is called variable. It is also known as Identifier].

Rules for naming a variable:-

It can be of letters, digits and underscore(_)

First letter should be a letter or an underscore, but it should not be a digit.

Reserve words cannot be used as variable names.

Example: basic, root, rate, roll-no etc are valid names.

Declaration of variables:

Syntax	type	Variable list
int	i, j	i, j are declared as integers
float	salary	salary is declared ad floating point variable
Char	sex	sex is declared as character variable

2.4 Operators

An Operator is a symbol that operates on a certain data type. The data items that operators act upon are called **operands**. Some operators require two operands, some operators act upon only one operand. In C, operators can be classified into various categories based on their utility and action.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operator
4. Assignment Operator
5. Increment & Decrement Operator
6. Conditional Operator
7. Bitwise Operator
8. Comma Operator

1. Arithmetic Operators

The Arithmetic operators performs arithmetic operations. The Arithmetic operators can operate on any built in data type. A list of arithmetic operators are

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division

2. Relational Operators

Relational Operators are used to compare arithmetic, logical and character expressions. The Relational Operators compare their left hand side expression with their right hand side expression. Then evaluates to an integer. If the Expression is false it evaluate to “zero”(0) if the expression is true it evaluate to “one”

Operator	**Meaning**
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greater than or Equal to
= =	Equal to

The Relational Operators are represented in the following manner:

Expression-1 Relational Operator Expression-2

The Expression-1 will be compared with Expression -2 and depending on the relation the result will be either "TRUE" OR "FALSE".

Examples :

Expression Evaluate to

(5 <= 10) ———————— 1

(-35 > 10) ———————— 0

(X < 10) ———————— 1 (if value of x is less than 10)

0 Other wise

(a + b) = = (c + d) 1 (if sum of a and b is equal to sum of c, d)

0 Other wise

3. Logical Operators

A logical operator is used to evaluate logical and relational expressions. The logical operators act upon operands that are themselves logical expressions. There are three logical operators.

Operators	**Expression**
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical And (&&): A compound Expression is true when two expressionwhen two expressions are true. The && is used in the following manner.

Exp1 && Exp2.

The result of a logical AND operation will be true only if both operands are true.

The results of logical operators are:

Exp1 Op. Exp2 Result

True && True True
True && False False

False && False False

False && True False

Example: a = 5; b = 10; c = 15;

Exp1 Exp2 Result

1. (a< b) && (b < c) => True
2. (a> b) && (b < c) => False
3. (a< b) && (b > c) => False
4. (a> b) && (b > c) => False

Logical OR*:* A compound expression is false when all expression arefalse otherwise the compound expression is true. The operator "||" is used as It evaluates to true if either exp-1 or exp-2 is true. The truth table of "OR" is Exp1 || Exp2

Exp1 Operator Exp2 Result:

Exp1	Operator	Exp2 Result
True	||	True True
True	||	False True
False	||	True True
False	||	False False

Example: a = 5; b = 10; c = 15;

Exp1 Exp2 Result

1. (a< b) || (b < c) => True
2. (a> b) || (b < c) => True
3. (a< b) || (b > c) => True
4. (a> b) || (b > c) => False

Logical NOT: The**NOT (!)**operator takes single expression andevaluates to true(1) if the expression is false (0) or it evaluates to false (0) if expression is true (1). The general form of the expression.

! (Relational Expression)

The truth table of **NOT :**

Operator. Exp1 Result

! True False

! False True

Example: a = 5; b = 10; c = 15

1. !(a< b) False

2. !(a> b) True

4. Assignment Operator

An assignment operator is used to assign a value to a variable. The most commonly used assignment operator is =. The general format for assignment operator is :

<Identifer> = < expression >

Where identifier represent a variable and expression represents a constant, a variable or a Complex expression.

If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left.

Example: Suppose that **I** is an **Integer** type Variable then

1. I = 3.3 3 (Value of I)

2. I = 3.9 3 (Value of I)

3. I = 5.74 5 (Value of I)

Multiple assignment

< identifier-1 > = < identifier-2 > = - - - = < identifier-n > = <exp>;

Example: a,b,c are integers; j is float variable

1. a = b = c = 3;

2. a = j = 5.6; then a = 5 and j value will be 5.6

C contains the following five additional assignment operators

1. += **2.**-= **3.** += **4.** *= **5.** /=

The assignment expression is: - Exp1 < Operator> Exp-2

Ex: I = 10 (assume that)

Expression Equivalent to Final Value of 'I'

1. I + = 5 I = I + 5 15

2. I - = 5 I = I - 5 10

3. I * = 5 I = I * 5 50

4. I / = 5 I = I / 5 10

5. Increment & Decrement Operator

The increment/decrement operator act upon a Single operand and produce a new value is also called as "**unary operator"**. The increment operator ++ adds 1 to the operand and the Decrement operator – subtracts 1 from the operand.

Syntax: < operator >< variable name >;

The ++ or – operator can be used in the two ways.

Example : ++ a; Pre-increment (or) a++ Post increment —a; Pre-Decrement (or) a— Post decrement

1. ++ a Immediately increments the value of a by 1.

2. a ++ The value of the a will be increment by 1 after it is utilized.

Example 1*:* Suppose a = 5 ;

Statements Output

printf ("a value is %d", a); a value is 5

printf ("a value is %d", ++ a); a value is 6

printf ("a value is %d ", a) ; a value is 6

Example 2*:*Suppose : a = 5 ;

Statements Output

printf ("a value is %d ", a); a value is 5

printf ("a value is %d ", a++); a value is 5

printf ("a value is %d ",a); a value is 6

a and a- will be act on operand by decrement value like increment operator.

6. Conditional operator (or) Ternary operator (? :)

It is called ternary because it uses three expression. The ternary operator acts like If- Else construction.

Syn :(<Exp –1 > ?<Exp-2> : <Exp-3>);

Expression-1 is evaluated first. If Exp-1 is true then Exp-2 is evaluated other wise it evaluate Exp-3 will be evaluated.

Flow Chart :

Exp-1

Exp-2 Exp-3

Exit

Example:

1. a = 5 ; b = 3;

(a> b ? printf (“a is larger”) : printf (“b is larger”));

Output is :a is larger

2. a = 3; b = 3;

(a>b ?printf (“a is larger”) : printf (“b is larger”));

Output is :b is larger

7. Bit wise Operator

A bitwise operator operates on each bit of data. These bitwiseoperator can be divided into three categories.

i. The logical bitwise operators.
ii. The shift operators
iii. The one’s complement operator.

i) The logical Bitwise Operator : There are three logical bitwise operators.

a) Bitwise **AND &**

b) Bitwise **OR |**

c) Bitwise exclusive **XOR ^**

Suppose b1 and b2 represent the corresponding bits with in the first and second operands, respectively.

B1 B2 B1 & B2 B1 | B2 B1 ^ B2

1 1 1 1 0

1 0 0 1 1

0 1 0 1 1

0 0 0 0 0

The operations are carried out independently on each pair of corresponding bits within the operand thus the least significant bits (ie the right most bits) within the two operands. Will be compared until all the bits have been compared. The results of these comparisons are

A **Bitwise AND** expression will return a 1 if both bits have a value of 1. Other wise, it will return a value of 0.

A **Bitwise OR** expression will return a 1 if one or more of the bits have a value of 1. Otherwise, it will return a value of 0.

A **Bitwise EXCLUSIVE OR** expression will return a 1 if one of the bits has a value of 1 and the other has a value of 0. Otherwise, if will return a value of 0.

Example::**Variable Value Binary Pattern**

X 5 0101

Y 2 0010

X & Y 0 0000

X | Y 7 0111

X ^ Y 7 0111

ii) The Bitwise shift Operations: The two bitwise shift operators are **Shift left** (<<) and **Shift right** (>>). Each operator requires two operands.The first operand that represents the bit pattern to be shifted. The second is an unsigned integer that indicates the number of displacements.

Example: c = a << 3;

The value in the integer a is shifted to the left by three bit position. The result is assigned to the c.

A = 13; c= A<<3;

Left shift << c= 13 * 2 3 = 104;

Binary no 0000 0000 0000 1101

After left bit shift by 3 places ie,. a<<3

0000 0000 0110 1000

The right –bit – shift operator (>>) is also a binary operator.

Example: c = a >>2 ;

The value of a is shifted to the right by 2 position

insert 0's Right – shift >> drop off 0's

0000 0000 0000 1101

After right shift by 2 places is a>>2

0000 0000 0000 0011 c=13>>2

c= 13/4=3

iii) Bit wise complement: The complement op.~ switches all the bits in abinary pattern, that is all the 0's becomes 1's and all the 1's becomes 0's.

variable value Binary patter

x 23 0001 0111

~x 132 1110 1000

8. Comma Operator

A set of expressions separated by using commas is a valid construction in c language.

Example :inti, j;

i= (j = 3, j + 2) ;

The first expression is j = 3 and second is j + 2. These expressions are evaluated from left to right. From the above example I = 5.

Size of operator: The operator size operator gives the size of the datatype or variable in terms of bytes occupied in the memory. This operator allows a determination of the no of bytes allocated to various Data items

Example :inti; float x; double d; char c;**OUTPUT**

```
Printf ("integer : %d\n", sizeof(i)); Integer : 2
Printf ("float : %d\n", sizeof(i)); Float : 4
Printf ("double : %d\n", sizeof(i)); double : 8
Printf ("char : %d\n",sizeof(i)); character : 1
```

2.5 Expressions

An expression can be defined as collection of data object and operators that can be evaluated to lead a single new data object. A data object is a constant, variable or another data object.

Example : a + b

x + y + 6.0

3.14 * r * r

(a + b) * (a – b)

The above expressions are called as arithmetic expressions because the data objects (constants and variables) are connected using arithmetic operators.

Evaluation Procedure: The evaluation of arithmetic expressions is as perthe hierarchy rules governed by the C compiler. The precedence or hierarchy rules for arithmetic expressions are

1. The expression is scanned from left to right.
2. While scanning the expression, the evaluation preference for the operators are

*, /, % - evaluated first

+, - - evaluated next

3. To overcome the above precedence rules, user has to make use of parenthesis. If parenthesis is used, the expression/ expressions with in parenthesis are evaluated first as per the above hierarchy.

UNIT - II

Statements

Data Input & Output

An input/output function can be accessed from anywhere within a program simply by writing the function name followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function.
Some input/output Functions do not require arguments though the empty parentheses must still appear. They are:

	Input Statements	*Output Statements*
Formatted	scanf()	printf()
Unformatted	getchar()gets()	putchar()puts()

getchar()

Single characters can be entered into the computer using the C library Function **getchar()**. It returns a single character from a standard input device. The function does not require any arguments.

Syntax: <Character variable> = getchar();

Example: char c;

c = getchar();

putchar()

Single characters can be displayed using function **putchar()**. It returns a single character to a standard output device. It must be expressed as an argument to the function.

Syntax: putchar(<character variable>);

Example: char c;

____

putchar(c);

gets()

The function **gets()** receives the string from the standard input device.

Syntax: gets(<string type variable or array of char>);

Where s is a string.

The function gets accepts the string as a parameter from the keyboard, till a newline character is encountered. At end the function appends a “null” terminator and returns.

puts()

The function **puts()** outputs the string to the standard output device.

Syntax: puts(s);

Where s is a string that was real with gets();

Example:

```
main()
{
char line[80];
gets(line);
puts(line);
}
```

scanf()

Scanf() function can be used input the data into the memory from the standard input device. This function can be used to enter any combination of numerical Values, single characters and strings. The function returns number of data items.

Syntax:-scanf (“control strings”, &arg1,&arg2,——&argn);

Where control string referes to a string containing certain required formatting information and arg1, arg2——argn are arguments that represent the individual input data items.

Example*:*

```
#include<stdio.h>
main()
{
char item[20];
intpartno;
float cost;
scanf(“%s %d %f”,&item,&partno,&cost);
}
```

Where s, d, f with % are conversion characters. The conversion characters indicate the type of the corresponding data. Commonly used conversion characters from data input.

Conversion Characters

Characters	Meaning
%c	data item is a single character.
%d	data item is a decimal integer.
%f	data item is a floating point value.
%e	data item is a floating point value.
%g	data item is a floating point value.
%h	data item is a short integer.
%s	data item is a string.
%x	data item is a hexadecimal integer.
%o	data item is a octal interger.

printf()

The printf() function is used to print the data from the computer's memory onto a standard output device. This function can be used to output any combination of numerical values, single character and strings.

Syntax: printf("control string", arg-1, arg-2,———arg-n);

Where control string is a string that contains formatted information, and arg-1, arg-2 —— are arguments that represent the output data items.

Example:

```
#include<stdio.h>
main()
{
char item[20];
intpartno;
float cost;
_______________
printf ("%s %d %f", item, partno, cost);
}                    (Where %s %d %f are conversion characters.)
```

2.6 Assignment Statement

Assignment statement can be defined as the statement through which the value obtained from an expression can be stored in a variable.

The general form of assignment statement is

< variable name> = < arithmetic expression> ; Example: sum = a + b + c;

tot = s1 + s2 + s3;

area = ½ * b* h;

2.7 I/O Control Structure (if, If-else, for, while, do-while)

Conditional Statements

The conditional expressions are mainly used for decision making. The following statements are used to perform the task of the conditional operations.

a. if statement.

b. If-else statement. Or 2 way if statement

c. Nested else-if statement.

d. Nested if –else statement.

e. Switch statement.

a. if statement

The **if statement** is used to express conditional expressions. If the given condition is true then it will execute the statements otherwise skip the statements.

The simple structure of '**if**' statement is

i. If (<condtionalexpressione>)

statement-1;

(or)

ii. If (<condtionalexpressione>)

{

```
statement-1;
statement-2;
statement-3;
……………
……………
STATEMENT-N
}
```

The expression is evaluated and if the expression is true the statements will be executed. If the expression is false the statements are skipped and execution continues with the next statements.

Example: a=20; b=10;

```
if ( a > b )
printf (“big number is %d” a);
```

b. if-else statements

The **if-else** statements is used to execute the either of the two statements depending upon the value of the exp. The general form is

```
if(<exp>)
{
Statement-1;
Statement -2;
…………..          “ SET-I”
……………
Statement- n;
}
else
{
Statement1;
Statement 2;
```

………….. " SET-II

……………

Statement n;

}

SET - I Statements will be executed if the exp is true.

SET – II Statements will be executed if the exp is false.

Example:

```
if( a> b )
printf ("a is greater than b");
else
printf ("a is not greater than b");
```

c. Nested else-if statements

If some situations if may be desired to nest multiple **if-else** statements. In this situation one of several different course of action will be selected.

Syntax

```
if( <exp1> )
        Statement-1;
else if ( <exp2> )
        Statement-2;
else if ( <exp3> )
        Statement-3;
else
        Statement-4;
```

When a logical expression is encountered whose value is true the corresponding statements will be executed and the remainder of the nested else if statement will be bypassed. Thus control will be transferred out of the entire nest once a true condition is encountered.

The final **else** clause will be apply if none of the exp is true.

d. nestedif-else statement

It is possible to nest if-else statements, one within another. There are several different form that nested if-else statements can take.

The most general form of two-layer nesting is

```
if(exp1)
        if(exp3)
    Statement-3;
        else
    Statement-4;
else
        if(exp2)
    Statement-1;
        else
    Statement-2;
```

One complete **if-else** statement will be executed if **expression1** is true and another complete **if-else** statement will be executed if **expression1** is false.

e. Switch statement

A switch statement is used to choose a statement (for a group of statement) among several alternatives. The switch statements is useful when a variable is to be compared with different constants and in case it is equal to a constant a set of statements are to be executed.

Syntax:

```
Switch (exp)
{
case
    constant-1:
    statements1;
case
    constant-2:
```

```
        statements2;
        ______
        ______
    default:
        statement n;
    }
```

Where constant1, constanat2 — — — are either integer constants or character constants. When the switch statement is executed the exp is evaluated and control is transferred directly to the group of statement whose case label value matches the value of the exp. If none of the case label values matches to the value of the exp then the default part statements will be executed.

If none of the case labels matches to the value of the exp and the default group is not present then no action will be taken by the switch statement and control will be transferred out of the switch statement.

A simple switch statement is illustrated below.

Example 1:

```
main()
{
char choice;
printf("Enter Your Color (Red - R/r, White – W/w)");
choice=getchar();
switch(choice= getchar())
{
case'r':
case'R':
printf ("Red");
break;
case'w':
case'W':
```

```
printf ("white");
break;
default :
printf ("no colour");
}
```

Example 2:

```
switch(day)
{
case 1:
printf ("Monday");
break;
______
______
}
```

2.8 Structure for Looping Statements

Loop statements are used to execute the statements repeatedly as long as an expression is true. When the expression becomes false then the control transferred out of the loop. There are three kinds of loops in **C**.

a) while   b) do-while   c) for

a. while statement

while loop will be executed as long as the exp is true.

Syntax:

```
while (exp)
{
statements;
}
```

The statements will be executed repeatedly as long as the exp is true. If the exp is false then the control is transferred out of the while loop.

Example:

```
int digit = 1;
While (digit <=5) FALSE
```

{

printf (“%d”, digit); TRUE

Cond Exp

Statements; ++digit;

}

The while loop is top tested i.e., it evaluates the condition before executing statements in the body. Then it is called entry control loop.

b. do-while statement

The **do-while** loop evaluates the condition after the execution of the statements in the body.

Syntax: do

Statement;

While<exp>;

Here also the statements will be executed as long as the exp value is true. If the expression is false the control come out of the loop.

Example:

-int d=1;

do

{

printf (“%d”, d); FALSE

++d;

} **while** (d<=5);

TRUE Cond Exp

statements

exit

The statement with in the do-while loop will be executed at least once. So the **do-while** loop is called a bottom tested loop.

c. for statement

The **for** loop is used to executing the structure number of times. The **for** loop includes three expressions. First expression specifies an initial value for an index (initial value), second expression that determines whether or not the loop is continued (conditional statement) and a third expression used to modify the index (increment or decrement) of each pass.

Note: Generally for loop used when the number of passes is known inadvance.

Syntax: **for** (exp1;exp2;exp3)

```
{
        Statement –1;
        Statement – 2;
        ——————; FALSE
        ——————;
        Statement - n; TRUE
}
exp2
        Statements;
exp3
Exit loop
exp1
start
```

Where **expression-1** is used to initialize the control variable. This expression is executed this expression is executed is only once at the time of beginning of loop.

Where **expression-2** is a logical expression. If **expression-2** is true, the statements will be executed, other wise the loop will be terminated. This expression is evaluated before every execution of the statement.

Where **expression-3** is an increment or decrement expression after executing the statements, the control is transferred back to the

expression-3 and updated. There are different formats available in **for loop**. Some of the expression of loop can be omit.

Format - I

```
for( ; exp2; exp3 )
Statements;
```

In this format the initialization expression (i.e., **exp1**) is omitted. The initial value of the variable can be assigned outside of the **for loop.**

Example 1

```
inti = 1;
for( ; i<=10; i++ )
printf (“%d \n”, i);
```

Format - II

```
for( ; exp2 ; )
Statements;
```

In this format the initialization and increment or decrement expression (i.e**expression-1** and **expression-3**) are omitted. The exp-3 can be given at thestatement part.

Example 2

```
inti = 1;
for( ; i<=10; )
{
printf (“%d \n”,i);
i++;
}
```

Formate - III

```
for( ; ; )
Statements;
```

In this format the **three expressions** are omitted. The loop itself assumes the **expression-2**is true. So **Statements** will be executed infinitely.

Example 3

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

2.9 Nested Looping Statements

Many applications require nesting of the loop statements, allowing on loop statement to be embedded with in another loop statement.

Definition

Nesting can be defined as the method of embedding one control structure with in another control structure.

While making control structure s to be reside one with in another ,the inner and outer control structures may be of the same type or may not be of same type. But ,it is essential for us to ensure that one control structure is completely embedded within another.

```
/*program to implement nesting*/
#include <stdio.h>
main()
{
inta,b,c,
for (a=1,a< 2, a++)
{
printf ("%d",a)
for (b=1,b<=2,b++)
{
```

```
print f(%d",b)
for (c=1,c<=2,c++)
{
print f( " My Name is Sunny \n");
                    }
            }
     }
}
```

2.10 Multi Branching Statement (switch), Break, and Continue

For effective handling of the loop structures, C allows the following types of control break statements.

a. Break Statement b. Continue Statement

a. Break Statement

The break statement is used to terminate the control form the loops or to exit from a switch. It can be used within a for, **while, do-while, for**.

The general format is :

```
break;
```

If **break** statement is included in a **while, do-while** or **for** then control will immediately be transferred out of the loop when the break statement is encountered.

Example

```
for( ; ; ) normal loop
{
break
Condition
within loop
scanf ("%d",&n);
if ( n < -1)
```

```
break;
sum = sum + n;
}
```

b. The Continue Statement

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Rather, the remaining loop statements are skipped and the proceeds directly to the next pass through the loop. The “**continue**” that can be included with in a **while a do-while and a for loop** statement.

General form :

continue;

The **continue** statement is used for the inverse operation of the **break**statement .

Condition

with in loop

Remaining part of loop

continue

Example

```
while (x<=100)
{
if (x <= 0)
{
printf (“zero or negative value found \n”);
continue;
}
}
```

The above program segment will process only the positive whenever a zero or negative value is encountered, the message will be displayed and it continue the same loop as long as the given condition is satisfied.

2.12 Unconditional Branching (Go To Statement)

goto statement

The **go to statement** is used to alter the program execution sequence by transferring the control to some other part of the program.

Syntax

Where label is an identifier used to label the target statement to which the control would be transferred the target statement will appear as:

Syntax

```
goto<label>;
label :
statements;
```

Example 1

```
#include <stdio.h>
main();
{
inta,b;
printf (“Enter the two numbers”);
scanf (“%d %d”,&a,&b);
if (a>b)
goto big;
else
goto small;
big:printf (“big value is %d”,a);
goto stop;
small:printf (“small value is %d”,b);
goto stop;
stop;
}
```

Functions

3.0 Introduction

Experienced programmer used to divide large (lengthy) programs in to parts, and then manage those parts to be solved one by one. This method of programming approach is to organize the typical work in a systematic manner. This aspect is practically achieved n C language thorough the concept known as ‘Modular Programming”.

The entire program is divided into a series of modules and each module is intended to perform a particular task. The detailed work to be solved by the module is described in the module (sub program) only and the main program only contains a series of modulus that are to be executed. Division of a main program in to set of modules and assigning various tasks to each module depends on the programmer’s efficiency.

Whereas there is a need for us repeatedly execute one block of statements in one place of the program, loop statements can be used. But, a block of statements need to be repeatedly executed in many parts of the program, then repeated coding as well as wastage of the vital computer resource memory will wasted. . If we adopt modular programming technique, these disadvantages can be eliminated. The modules incorporated in C are called as the FUNCTIONS, and each function in the program is meant for doing specific task. C functions are easy to use and very efficient also.

3.1 Functions

Definition

A function can be defined as a subprogram which is meant for doing a specific task.

In a C program, a function definition will have name, parentheses pair contain zero or more parameters and a body. The parameters used in the parenthesis need to be declared with type and if not declared, they will be considered as of integer type.

The general form of the function is :

```
function type name <arg1,arg2,arg3, ————,argn>)
  data type arg1, arg2,;
```

```
data type argn;
  {
        body of function;

        ______________________
        ______________________
        ______________________

        return (<something>);
}
```

From the above form the main components of function are

- Return type
- Function name
- Function body
- Return statement

Return Type

Refers to the type of value it would return to the calling portion of the program. It can have any of the basic data types such as int, float, char, etc. When a function is not supposed to return any value, it may be declared as type void

Example

```
void function name(- - - - - - - - - -);
int function name( - - - - - - - - - - - );
char function name ( — - - - - - - );
```

Function Name

The function name can be any name conforming to the syntax rules of the variable.

A function name is relevant to the function operation.

Example

```
output( );
read data( );
```

Formal arguments

The arguments are called **formal arguments (or) formal parameters**, because they represent the names of data items that are transferred into the function from the calling portion of the program.

Any variable declared in the body of a function is said to be local to that function, other variable which were not declared either arguments or in the function body, are considered "**globol"** to the function and must be defined externally.

Example

```
int biggest (int a, int b)
{
________________________
________________________
________________________
return( );
}
```

a, b are the formal arguments.

Function Body

Function body is a compound statement defines the action to be taken by the function. It should include one or more **"return"** statement in order to return a value to the calling portion of the program.

Example

```
int biggest(int a, int b)
{
if ( a > b)
return(a); body of function.
else
return(b);
}
```

Every C program consists of one or more functions. One of these functions must be called as **main.** Execution of the program will always begin by carrying out the instructions in main. Additional functions will be subordinate to main. If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition can't be embedded within another.

Generally a function will process information that is passed to it from the calling portion of the program and return a single value. Information is passed to the function via arguments (**parameters**) and returned via the **"return"** statement.

Some functions accept information but do not return anything (***ex:***printf()) whereas other functions (***ex:***scanf()) return multiple values.

3.1.1 The Return Statement

Every function subprogram in C will have return statement. This statement is used in function subprograms to return a value to the calling program/function. This statement can appear anywhere within a function body and we may have more than one return statement inside a function.

The general format of return statement is

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

If no value is returned from function to the calling program, then there is no need of return statement to be present inside the function.

Programs using function Call Techniques

Example 1: Write a program to find factorial to the given positive integer,using function technique.

```
# include
<stdio.h> main( )
{
    int n;
    printf ( " Enter any positive
    number\n"); scanf( "%d", &n);
```

```
printf( " The factorial of %d s %d \n",fact (n));
}
fact(i)
int I;
{
    int j; f = 1 ;
    for ( j = I; j>0; j - -)
    f  = f * I;
    return ( f ) ;
}
```

In the above program function with name 'fact' is called by the main program. The function fact is called with n as parameter. The value is returned through variable f to the main program.

Example 2: Write a program to find the value of f(x) as $f(x) = x^2 + 4$, forthe given of x. Make use of function technique.

```
# include
<stdio.h> main( )
{
      f ( );
}
f ( )
 { intx,y ;
 printf( " Enter value of x \n");
 scanf( " %d", & x );
  y = (x * x + 4);
 printf ( " The value of f (x) id %d \n", y ) ;
}
```

3.3 Advantages of Function

The main advantages of using a function are:

- Easy to write a correct small function
- Easy to read and debug a function.
- Easier to maintain or modify such a function
- Small functions tend to be self documenting and highly readable
- It can be called any number of times in any place with different parameters.

Storage class

A variable's storage class explains where the variable will be stored, its initial value and life of the variable.

Iteration

The block of statements is executed repeatedly using loops is called Iteration

Categories of Functions

A function, depending on, whether arguments are present or not and a value is returned or not.

A function may be belonging to one of the following types.

1. Function with no arguments and no return values.
2. Function with arguments and no return values.
3. Function with arguments and return values

3.4 Advanced Featured of Functions

a. Function Prototypes

b. Calling functions by value or by reference

c. Recursion.

a. Function Prototypes

The user defined functions may be classified as three ways based on the formal arguments passed and the usage of the **return** statement.

a. Functions with no arguments and no return value

b. Functions with arguments no return value

c. Functions with arguments and return value.

a. Functions with no arguments and no return value

A function is invoked without passing any formal arguments from the calling portion of a program and also the function does not return back any value to the called function. There is no communication between the calling portion of a program and a called function block.

Example:

```
#include <stdio.h>
main()
{
        void message( ); Function declaration
        message( ); Function calling
}
void message( )
{
        printf ("GOVT JUNIOR COLLEGE \n");
        printf ("\t HYDERABAD");
}
```

b. Function with arguments and no return value

This type of functions passes some formal arguments to a function but the function does not return back any value to the caller. It is any one way data communication between a calling portion of the program and the function block.

Example

```
#include <stdio.h>
main()
{
        void square(int);
        printf ("Enter a value for n \n");
        scanf ("%d",&n);
        square(n);
}
```

```
void square (int n)
{
        int value;
        value = n * n;
        printf ("square of %d is %d ",n,value);
}
```

c. Function with arguments and return value

The third type of function passes some formal arguments to a function from a calling portion of the program and the computer value is transferred back to the caller. Data are communicated between the calling portion and the function block.

Example

```
#include <stdio.h>
main()
{
        int square (int);
        int value;
        printf ("enter a value for n \n");
        scanf("%d", &n);
        value = square(n);
        printf ("square of %d is %d ",n, value);
}
int square(int n)
{
        int p;
        p = n * n;
        return(p);
}
```

The keyword **VOID** can be used as a type specifier when defining a function that does not return anything or when the function definition does not include any arguments.

The presence of this keyword is not mandatory but it is good programming practice to make use of this feature.

Actual and Formal Parameters (or) Arguments

Function parameters are the means of communication between the calling and the called functions. The parameters may classify under two groups.

1. Formal Parameters
2. Actual Parameters

1. Formal Parameters

The formal parameters are the parameters given in function declaration and function definition. When the function is invoked, the formal parameters are replaced by the actual parameters.

2. Actual Parameters

The parameters appearing in the function call are referred to as actual parameters. The actual arguments may be expressed as constants, single variables or more complex expression. Each actual parameter must be of the same data type as its corresponding formal parameters.

Example

```
#include <stdio.h>
int sum (int a , int b )
{
        int c;
        c = a + b;
        return(c);
}
main( )
{
        intx,y,z;
        printf (“enter value for x,y \n”);
        scanf (“%d %d”,&x,&y);
        z = x + y;
        printf (“ sum is = %d”,z);
}
```

The variables **a and b** defined in function definition are known as **formalparameters**. The variables **x and y** are **actual parameters**.

Local and Global Variable:

The variables may be classified as local or global variables.

Local Variable

The variables defined can be accessed only within the block in which they are declared. These variables are called "Local" variables

Example

```
funct (int ,int j)
{
        intk,m;
        ________;
        ________;
}
```

The integer variables **k** and **m** are defined within a function block of the "**funct()**". All the variables to be used within a function block must be either defined at the beginning of the block or before using in the statement. Local variables one referred only the particular part of a block of a function.

Global Variable

Global variables defined outside the main function block. Global variables are not contained to a single function. Global variables that are recognized in two or more functions. Their scope extends from the point of definition through the remainder of the program.

b. Calling functions by value or by reference

The arguments are sent to the functions and their values are copied in the corresponding function. This is a sort of information inter change between the calling function and called function. This is known as Parameter passing. It is a mechanism through which arguments are passed to the called function for the required processing. There are two methods of parameter passing.

1. Call by Value
2. Call by reference.

1. **Call by value:** When the values of arguments are passed from callingfunction to a called function, these values are copied in to the called

function. If any changes are made to these values in the called function, there are NOCHANGE the original values within the calling function.

Example

```
#include <stdio.h>
main();
{
        int n1,n2,x;
        intcal_by_val();
        N1 = 6;
        N2 = 9;
        printf( n1 = %d and n2= %d\n", n1,n2);
        X = cal_by_Val(n1,n2);
        Printf( n1 = %d and n2= %d\n", n1,n2);
        Printf(" x= %d\n", x);
                / * end of main*/
        /*function to illustrate call by value*/
        Cal_by_val(p1,p2)
        int p1,p2;
        {
                int sum;
                Sum = (p1 + p2);
                P1 + = 2;
                P2* = p1;
                printf( p1 = %d and p2= %d\n", p1,p2);
                return( sum);
        }
}
```

When the program is executed the output will be displayed

N1 = 6 and n2 = 9

P1 = 8 and p2 = 72

N1 = 6 and n2 = 9

X = 15

There is NO CHANGE in the values of n1 and n2 before and after the function is executed.

2. Call by Reference: In this method, the actual values are not passed,instead their addresses are passed. There is no copying of values since their memory locations are referenced. If any modification is made to the values in the called function, then the original values get changed with in the calling function. Passing of addresses requires the knowledge of pointers.

Example

This program accepts a one-dimensional array of integers and sorts them in ascending order. [This program involves passing the array to the function].

```
#include
<stdio.h> main();
{
intnum[20], I,max;
void sort_nums();
printf( " enter the size of the
array"\n"); scanf("%d", &max);
for(i=0; i<max;I++)
sort_nums(num,max)      /* Function reference*/
printf("sorted numbers are as follows\n");
for(i=0; i<max;I++)
printf("%3d\n",num[i]);
/* end of the main*/
/* function to sort list of numbers*/
```

```
Void sort_nums(a,n)
Inta[],n;
{
        IntI,j,dummy;
        For(i=0;i<n;i++)
        {
                For(j=0; j<n; j++)
                {
                        If ( a[i] >a[j])
                        {
                                Dummy = a[i];
                                a[i] = a[j];
                                a[j] = dummy;
                        }
                }
        }
}
```

3.5 Recursion

One of the special features of C language is its support to recursion. Very few computer languages will support this feature.

Recursion can be defines as the process of a function by which it can call itself. The function which calls itself again and again either directly or indirectly is known as recursive function.

The normal function is usually called by the main ( ) function, by mans of its name. But, the recursive function will be called by itself depending on the condition satisfaction.

For Example,

```
main ( )
{
```

```
        f1( ) ;                    ——— Function called by main
      ____________
      ____________
        ____________
}
f1( ) ;                    ——— Function definition
{
      ____________
      ____________
      ____________
        f1( ) ;                    ——— Function called by itself
}
```

In the above, the main ( ) function s calling a function named f1( ) by invoking it with its name. But, inside the function definition f1( ), there is another invoking of function and it is the function f1( ) again.

Example programs on Recursion

Example 1 : Write a program to find the factorial of given non-negativeinteger using recursive function.

```
#include<stdio.h>
main ( )
{
        int result, n;
        printf( " Enter any non-negative integer\n");
        scanf ( " %d", & n);
        result = fact(n);
        printf ( " The factorial of %d is %d \n", n, result);
}
```

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

Example 2: Write 'C' program to generate Fibonacci series up to a limitusing recursion function. .

```
#include<stdio.h>
#include<conio.h>
int Fibonacci (int);
void main ( )
{
        inti, n;
        clrscr ( );
        printf ("Enter no. of Elements to be generated" \n)
        scanf ("%d", &n);
        for (i=1; i<n; i++)
        printf ("%d", Fibonacci (i));
        getch( );
}
```

```
int Fibonacci (int n)
{
        intfno;
        if (n= =1)
        return 1;
        else
            if (n= =2);
              return 1;
        else
             fno=Fibonacci (n-1) + Fibonacci (n-2);
        returnfno;
}
```

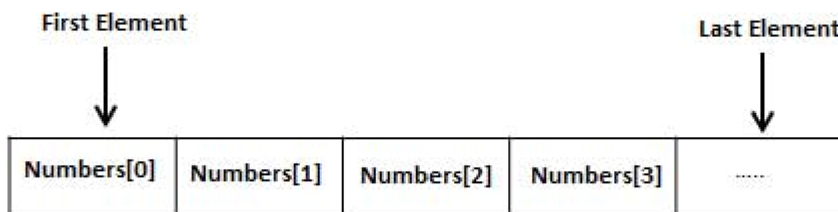
UNIT - III

4.0 Arrays

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

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

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



4.1 Declaring Arrays

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

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

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

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

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

4.2 Initializing Arrays

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

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

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

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

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

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

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

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

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

4.3 Accessing Array Elements

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

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

4.4. Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

4.4.1 Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

4.4.2 Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array.

5.0 Strings

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ −

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

```
#include <stdio.h>


int main () {


   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

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

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings −

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

6.0 Storage classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class 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 can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a

register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>


/* function declaration */

void func(void);


static int count = 5; /* global variable */


main() {


   while(count--) {

      func();

   }


   return 0;

}
```

```
/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

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

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

UNIT – IV

7.0 Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

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

- Title
- Author
- Subject
- Book ID

7.1 Defining a Structure

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

```
struct [structure tag] {


  member definition;

  member definition;

  ...

  member definition;

} [one or more structure variables];
```

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

```
struct Books {

  char  title[50];

  char  author[50];
```

```
  char  subject[100];
  int   book_id;
} book;
```

7.2 Accessing Structure Members

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

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

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


int main( ) {

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

  /* book 1 specification */
```

```
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

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

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

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

   return 0;
}
```

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

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

7.3 Bit Fields

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

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

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

```
struct packed_struct {

  unsigned int f1:1;

  unsigned int f2:1;

  unsigned int f3:1;

  unsigned int f4:1;

  unsigned int type:4;

  unsigned int my_int:9;

} pack;
```

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

7.4 Unoins

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

Defining a Union

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

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

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

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

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

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

```
#include <stdio.h>

#include <string.h>


union Data {

   int i;

   float f;

   char str[20];

};


int main( ) {


   union Data data;


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


   return 0;

}
```

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

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

Accessing Union Members

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

7.5 Pointers

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

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

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

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

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

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

7.5.1 Pointer Aarithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

```
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
```

```
  ptr = var;


  for ( i = 0; i < MAX; i++) {


    printf("Address of var[%d] = %x\n", i, ptr );

    printf("Value of var[%d] = %d\n", i, *ptr );


    /* move to the next location */

    ptr++;

  }


  return 0;

}
```

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

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

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

```
const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = &var[MAX-1];

   for ( i = MAX; i > 0; i--) {

      printf("Address of var[%d] = %x\n", i-1, ptr );
      printf("Value of var[%d] = %d\n", i-1, *ptr );

      /* move to the previous location */
      ptr--;
   }

   return 0;
}
```

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

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
```

```
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

7.5.2 Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int −

```
int **var;
```

UNIT – V

8.0 File Handling

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

calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

Opening Files

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

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

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

Sr.No.	Mode & Description
1	**r** Opens an existing text file for reading purpose.
2	**w** Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
3	**a** Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
4	**r+** Opens a text file for both reading and writing.
5	**w+** Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.

6	**a+** Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

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

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

Closing a File

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

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

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

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

Writing a File

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

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

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

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

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

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

```
#include <stdio.h>


main() {
  FILE *fp;


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

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

Reading a File

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

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

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

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

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

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

...) function to read strings from a file, but it stops reading after encountering the first space character.

```
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];

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

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

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

}
```

8.1 Memory Management

The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

Sr.No.	Function & Description
1	**void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will be **size**.
2	**void free(void *address);** This function releases a block of memory block specified by address.
3	**void *malloc(int num);** This function allocates an array of **num** bytes and leave them uninitialized.
4	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize**.

8.2 Command line Arguments

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

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

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

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


  if( argc == 2 ) {

    printf("The argument supplied is %s\n", argv[1]);

  }

  else if( argc > 2 ) {

    printf("Too many arguments supplied.\n");

  }

  else {

    printf("One argument expected.\n");

  }

}
```

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

```
$./a.out testing

The argument supplied is testing
```

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

```
$./a.out testing1 testing2

Too many arguments supplied.
```

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

```
$./a.out

One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''.

8.3 Preprocessor Directive

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef**

	Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the C to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

```
#include <stdio.h>

#include "myheader.h"
```

These directives tell the C to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h**from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE

#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE

  #define MESSAGE "You wish!"

#endif
```

It tells the C to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG

  /* Your debugging statements here */

#endif
```

It tells the C to process the statements enclosed if DEBUG is defined.

****