'C' LANGUAGE

HISTORY OF 'C' :

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972.It was designed and written by a man named Dennis Ritchie. It was first used as the systems language for the UNIX operating system. Ken Thompson, the developer of UNIX, had been using both an assembler and a language named B to produce initial versions of UNIX in 1970. C was invented to overcome the limitations of B.

The UNIX system is itself written in C. In fact C was invented specifically to implement UNIX. All of the UNIX commands which you type, plus the other system facilities such as password checking, line printer queues or magnetic tape controllers are written in C.

In 1960 there was computer languages come into existence specific purpose like FORTRAN for Scientific, COBOL for business applications. But people find it very difficult to learn many languages. So an international committee is formed to develop a language which suits all types of applications. The committees come up with a language called ALGOL 60.

ALGOL 60 never became popular because it seemed too abstract, too general. To reduce this abstractness and generality, a new language called Combined Programming Language (CPL) was developed at Cambridge University. But this turned out to be big, having so many feature, that it was hard to learn and implement. To overcome this, BCPL (Basic Combined Programming Language) was developed by Martin Richards at Cambridge University. This brings CPL down to its basic good features. But this was also too less powerful and too specific. Around same time Ken Thompson at AT & T's Bell Labs, as a further simplification of CPL wrote a language called B. But like BCPL B too turned out to be very specific. Ritchie inherited the features of B and BCPL, add some of his own and developed C.

Year	Language	Developed by	Remarks
1960	ALGOL	International Committee	Too general, too abstract.
1963	CPL	Cambridge University	Hard to learn, difficult to implement.
1967	BCPL	Martin Richards at Cambridge University Ken Thomson at AT & T	Could deal with only specific problems.
1970	B	Dennis Ritchie at AT & T	Could deal with only specific problems.
1972	C		Lost generality of BCPL and B restored.

FEATURES OF 'C':

- C is a general-purpose programming language.
- C is reliable, simple and easy to use.
- C is powerful because of its rich set of built-in-functions and operators.
- C is well suited for structured programming language bec' of its good control structures and modular structure.
- C is well suited for system programming because it is a middle level language bec' it has both features of assembly language and high level language.
- C programs are very fast due to its variety of data types.
- C is highly portable. i.e., programs written for one computer can run on any other machine.

Learning of any language begins at the character set that are available.

Character Set:

The characters set in C includes the following four categories:
1. Letters 2. Digits 3. Special characters 4. White spaces

Letters :
Uppercase A..Z
Lowercase a ..z

Digits:
All decimal digits 0..9

SPECIAL CHARACTERS

, comma
. period
; semicolon
: colon
? question mark
` apostrophe
" quotation mark
! exclamation mark
| vertical bar
/ slash
\ backslash
~ tild
_ under score
$ doller sign
% per cent sign
# number sign
& ampersand
^ caret
* asterisk
- minus
+ plus
< opening angle bracket(or less than sign)
> closing angle bracket(or greater than sign)
(left parenthesis
) right parenthesis
[left bracket
] right bracket
{ left brace
} right brace

White spaces
Blank spaces
Horizontal tab
Carriage return

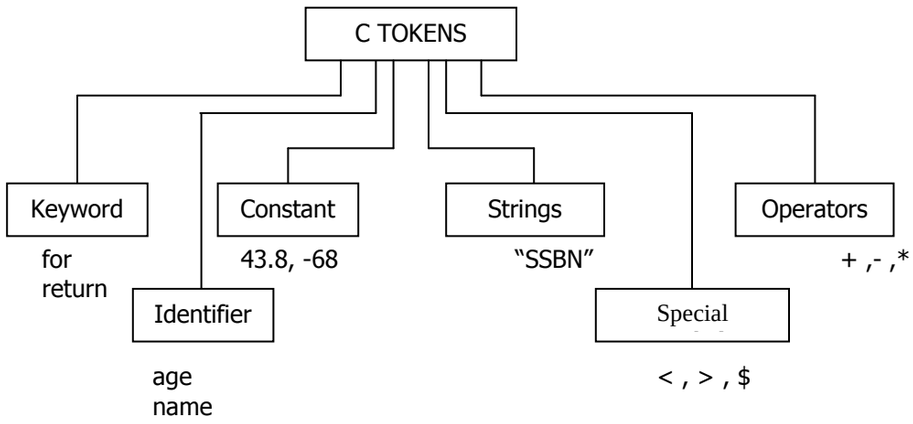
New line
Form feed

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

C Tokens :

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are knows as C tokens.

C has six types of tokens as shown in Fig. C programs are written using these tokens and the syntax of the language.



KEYWORDS AND IDENTIFIERS

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. All keywords must be written in lowercase.

KEYWORDS:

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

IDENTIFIERS:

Identifier is a user defined name that refers to the names of variables, constants, functions and arrays. These can also be termed as user-defined words. There are some rules for framing user-defined words. They are…

- The name should not exceed 8 characters.
- It can be a combination of Alphabets and digits.
- The first character must be an alphabet.
- No special characters are allowed other than _ (underscore).
- Both lower case and upper case characters are allowed, but it is case sensitive.

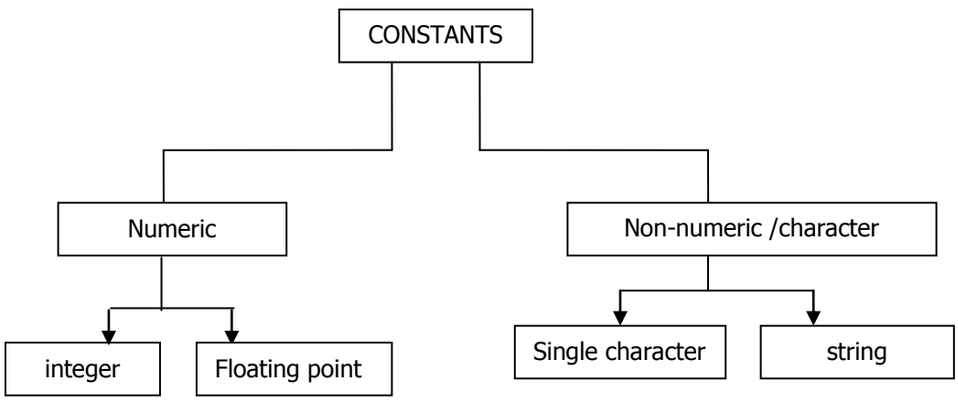
EXAMPLES: age, roll_no, name, no1 …

CONSTANTS: A constant is a fixed value that cannot be altered during the execution of a program

A Constant can be classified into two categories

1. Primary Constants
2. Secondary Constants

We would restrict our discussion to only primary constants.



C has FIVE types of primary constants. They are integer, float, char, logical, and string. The numeric constants can be preceded by a minus if needed.

INTEGER CONSTANTS:

An Integer constant is an integer-valued number consisting of a sequence of digits. The following are the rules for constructing integer constants.

(a). An integer constant must have at least one digit.

(b). It should not contain a decimal point.

(c). If a constant is positive ,it may or may not be preceded by a plus sign. If it is a negative, it must be preceded by a minus sign.

(d). Commas, blanks and non-digit characters are not allowed in integer constants.

(e). The value of integer constant cannot exceed specified limits.

The valid range is -32768 to +32767.(This range may larger for 32 bit computers).

EXAMPLES: Valid Integers: 12, 56, +456, -7865 …

Invalid Integer: 123.45, 45,356 …

NOTE: In C, Integer constants can also be written in other two number systems, Octal(base 8) and Hexadecimal (base 16).

An **octal integer** constant can consist of any combination of digits from the set 0 through 7.

The first digit must be 0 .

EXAMPLES: Valid Octal numbers 034, 0673, 0123

A hexadecimal integer constant must begin with either 0x OR 0X .

EXAMPLES : Valid hexadecimal numbers 0x12, 0XDF, 0x45E …

REAL CONSTANTS : Real constants are often called floating - point constants. There are two ways to represent a real constant decimal form and exponential form.

Real constants expressed in decimal form must have at least one digit and one decimal point. As in the case of integers, the real constants can be either +ve or -ve . Commas , blanks and non-digit characters are not allowed with in a real constant .

EXAMPLES : Acceptable real constants : .0 , 0.1 , 6.0 , -2000.0 , -0.000021 etc.,

In exponential form , a real constant is expressed as an integer number or a decimal number multiplied by an integral power of 10. This simplifies the writing of very large and very small numbers. In this representation, letter **e** is written instead of 10, the power is written just to the right of **e**. Generally , the part appearing before **e** is called **mantissa,** part following **e** is called exponent. An exponent part should be a whole number.

Rules for constructing real constants are :

(a) The mantissa part and the exponential part should be separated by a letter **e**.

(b) The mantissa part may have a +ve or -ve sign. Default sign of mantissa part is +ve.

(c) The exponent must have at least one digit which may have +ve or -ve sign.Default sign is +ve.

(d) Commas , blanks and non-digit characters are not allowed with in a real constant.

(e) The exponent part must be an integer.

(f) The mantissa part must have at least one digit.

(g) Range of real constants expressed in exponential form is -3.4e38 to +3.4e38.

EXAMPLES : Acceptable real constants in exponential form :

5.0**e**-12 , -7**e**+16 , 6.21**e**13 , -0.3**e**5

Notacceptable real constants in exponential form :

5.834**e**2.4 , **e** + 6

Character constants

A single character constant is a single character enclosed in single quotes.

EXAMPLES : ‘x’, ‘B’ ,‘9’ ,‘#’

Note that, the character constant '9' is not the same as the integer constant . The third constant in the above example is a blank space.

Character constants have integer values known as ASCII values.

String constants

A string constant is sequence of characters enclosed in double quotes. The characters may be any character from ‘C’ character set.

EXAMPLE : "SSBN Degree College"

"This is a string"

" a + b "

" "

"A"

Note : 1. " " is a **null string or empty string.**

2..The single string constant **"A"** is not equivalent to the single character constant **' A'** .

Backslash Character Constants:

C support special backslash character constants that are used in output functions. Note that each one of them represents one character, although they consist of two characters. These character combinations are Known as **escape sequences.**

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f '	form feed
'\n'	new line

'\r '	carriage return
'\t '	horizontal tab
'\v'	vertical tab
'\' '	single quote
'\" '	double quote
'\? '	question mark
'\\'	backslash
'\0'	null

Logical constants :

Though C does not provide logical datatype explicitly, the z**ero** and **non-zero** Values can be used as logical constants. Here , **0** means **false** and **non-zero** means true.

EXAMPLE : The constants 1, 50, -1 are considered to be true and 0 is considered as a logical constant false. These constants are extremely helpful in program where it is necessary to perform some action while a certain condition is true or false.

VARIABLES:

A variables is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.

Rules for constructing variable names:

- The name should not exceed 8 characters.
- It can be a combination of Alphabets and digits.
- The first character must be an alphabet.
- No special characters are allowed other than _ (underscore).
- Both lower case and upper case characters are allowed, but it is case sensitive.

EXAMPLES : Acceptable variable names : age, salary, p, dob_02.
Not Acceptable variable names : 563 , raj# , roll-no

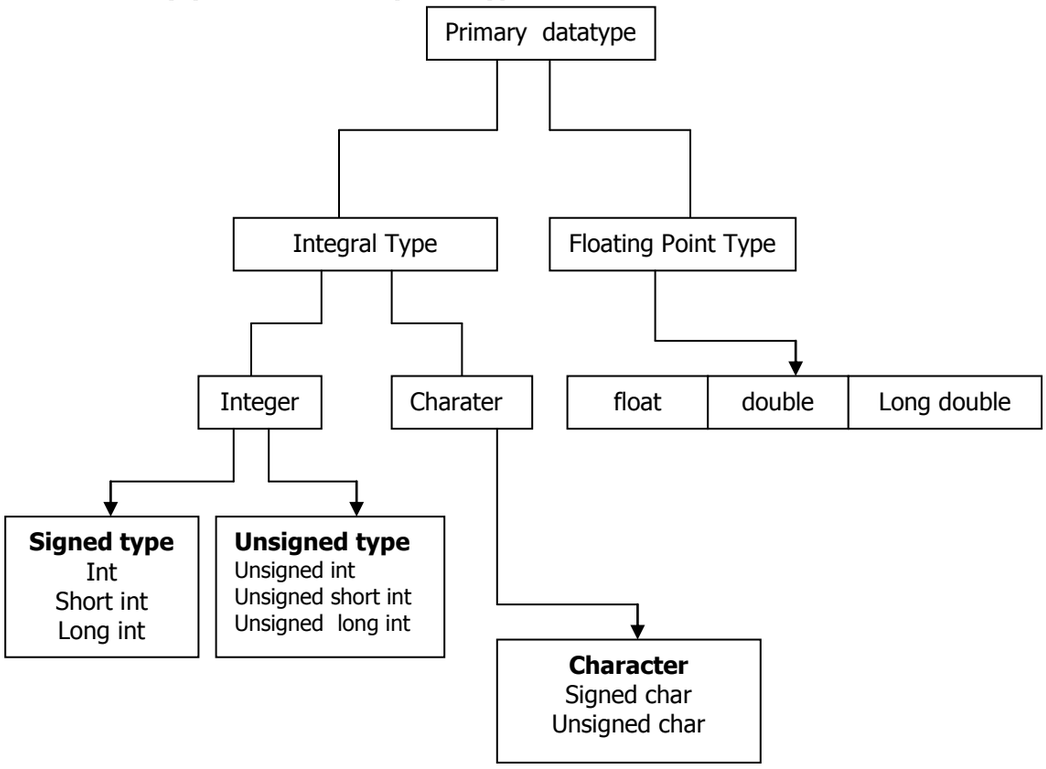
Type Declaration Instructions:

DATA TYPES: A data type is the type of the data item that is associated in the given application.
A data type defines a set of values and the operations that can be performed on them. Every data item in a C program has a data type associated with it.

C language is rich in its data types. **C** supports **four** classes of data types:

1) **Primary (or fundamental) data types** Ex: int, float, char
2) **Derived data types** Ex : arrays, functions, structures, and pointers
3) **User-defined data types** Ex : typedef , enum
4) **Empty data type** Ex: void

Primary (or fundamental) data type :



INTEGER Data Type : The whole numbers are known as Integers. In **C** the integer data can be stored in three ways. They are **short int, int, long int,** in both **signed and unsigned** forms. Unsigned is used if the sign has no significance so that to increase the range.

Data types and its ranges Table

Datatype	Description	Size(No.of Bytes)	Range
Sighed char	Character	1	-128 to 127
Unsigned char	Unsigned character	1	0 to 255
Signed int or int	Signed Integer	2	-32768 to +32767
Signed short int or short int	Short signed int	1	-128 to +127
Signed long int or long int	Long signed integer	4	-2,147,483,648 to 2,147,483,647
Unsigned int	unsigned Integer	2	0 to 65535
Unsigned short int	Unsigned short int	1	0 to 255
Unsigned long int	unsigned long int	4	0 to 4,294,967,295
Float	Floating point	4	-3.4 e 38 to1.7 e+308
Double	Double precision	8	1.7E -308 to 1.7E +308
Long double	Long double	10	-3.4 e-4932 to 1.1E+4932

Floating Point Type : Floating point type numbers are stored in 4 bytes(on all 16 bit and 32 bit machines), with 6 digits of precision. To increase the range of float data, the type **double** can be used to define the number.

A **double** data type numbers uses 8 bytes giving a precision of **14** digits. These are known as **double precision numbers**. Remember that double type represents the same data type that **float** represents, but with a great precision.

To extend the precision further, we may use **long double** which uses 10 bytes.

Character Type: A single character can be defined a character(char) type data.

Characters are usually stored in 8 bits of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to **char.**

DECLARATION OF VARIABLES: Declaration of a variable is used to specify the characteristics of a variable. I.e., name and type. This can be done by using Type Declaration Instruction. This is used to declare the type of variable being used in the program. Any variable used in the program must be declared before using it in any statement. The type declaration statement is usually written at the beginning of the C program.

The following syntax is used to declare the type of a variable.

Data type variable list;

Variables are separated by commas. A declaration statement must end with a semicolon.

Examples:

```
int i, j;
float marks, height;
```

User-Defined Type Declaration:

In C, the user-defined data type is of two types. They are

- Typedef
- Enumerated

In C there is a special feature called "type definition", using which the user can define an identifier as a new data type which represents the existing data type. This identifier can be used to declare the variables.

The general syntax is…

typedef type identifier

Where type refers to an existing data type and "identifier" refer to the "new " name given to the data type

Note: The typedef creates an alias to the existing data type but not a new type.

Examples: typedef int age;
typedef float height;

```
age stu1, stu2;
height h1, h2;
```

Enumerated data type:

The general syntax is as follows:

enum identifier {value1, value2,…valuen};

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces(known as enumeration constants).After this declaration , we can declare variables to be of this type as below:

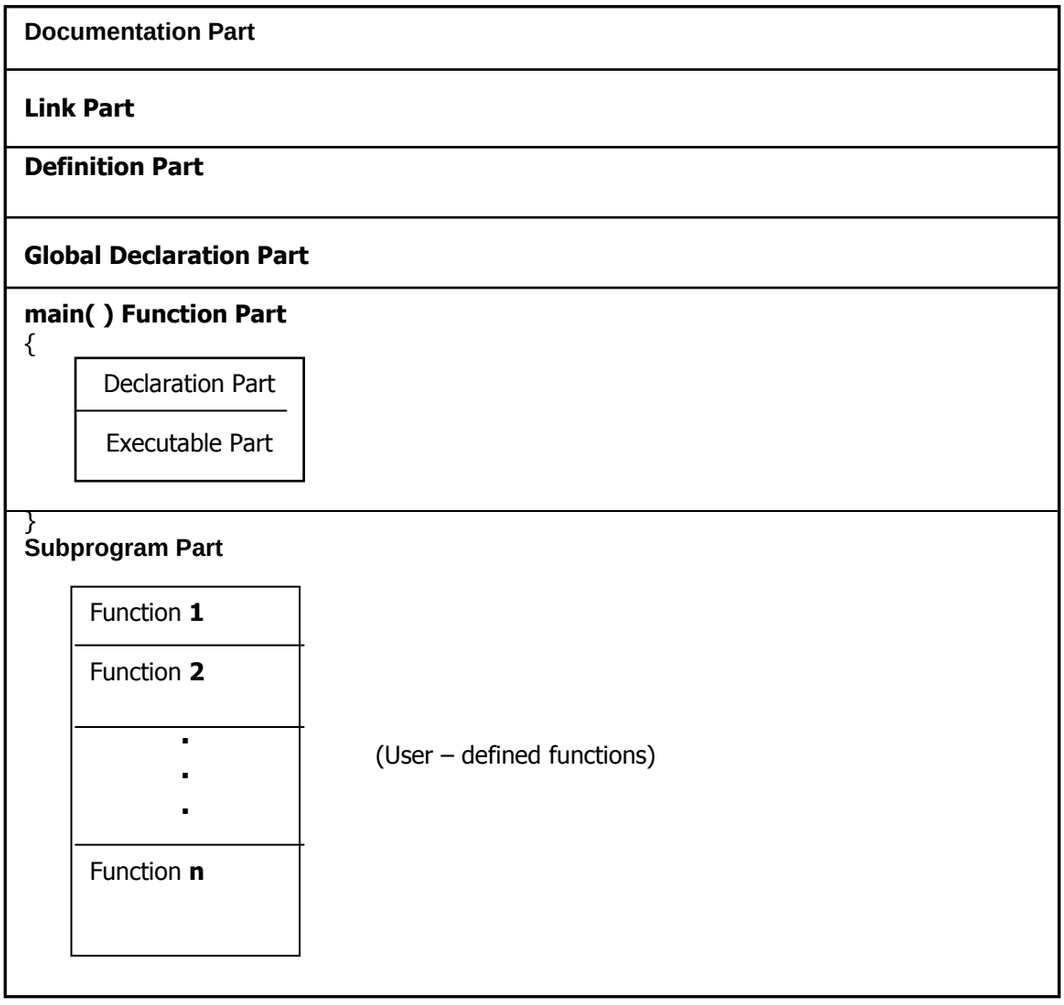
enum identifier ev1,ev2,..evn;

The enumerated variables ev1,ev2,..evn can only have one of the values in the given range.

Example:
enum month { January, February, March,. . . December};
enum month f-month, l-month;

BASIC STRUCTURE OF 'C' PROGRAMS

A 'C' program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. Write a 'C' program, we first create functions and then put them together. A 'C' program contain one or more sections shown in following figure.



Documentation Part:

This section includes the comment lines that are related to the program.

Link Part: This section includes the instructions that are needed to link the library functions to the program.
Definition Part: This section is used to define all the required constants and macros in the given program.
Global Declaration Part:

This section is used to declare the variables that are used in more than one function which are known as global variables. These are declared out side of all the functions.

Main Function Part:

This is must for any C program. It informs the system that the execution begins at this line. The main( ) is a special function used by the 'C' system to tell the computer where the program starts. Every program must have exactly one **main** function. This section is further divided into two parts. Declaration part and execution part. The first part declares all the variables that are used in the executable part. There must be atleast one statement in the second part. The two parts must be enclosed in parenthesis.

Sub program Part:

The **subprogram part** contains all the user-defined functions that are called in the main function.
User-defined functions are generally placed immediately after the main function, although they appear in any order.
All parts, except the main function part may be absent When they are not required.

SIMPLE 'C' PROGRAM

```
/*  Program to print a message  */
main( )
{
printf("WELCOME");

}
```

main() ← function name

```
{                ←————————start of program
…
…                }  program statements (executable and non-executable)
…
}                ←————————end of program
```

'C' Pre-processor: It is a program that processes our source program before it is passed to the compiler. In turbo C, **Ctrl F9** will do processing, compiling, linking as well as execution of the program is done automatically. The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. These can be placed anywhere in a program but are most often placed at the beginning of a program, before main(), or before beginning of a particular function.

The important pre-processor directives in C are

- Macro expansion
- File inclusion
- Conditional Compilation
- Miscellaneous directives

The simple directives are Macro expansion ---→ **#define**
and File inclusion-----→ **#include**

#include : This is used to include library files in to the program. The basic inclusion used is

#include<stdio.h>

This is used to make efficient use of I/O functions of the standard C library. When the processor encountered this directive, it includes the contents of the named file into the program exactly where the directive is written.

The filename is included in angle brackets to indicate it is located in a specified directory on the disk. If the header file resides in any other directory, the **#include** directive, indicating full path of that directory, can be written as **#include "stdio.h"**

#define: When the processor sees this directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. This is also used to define a string of characters that represents a numeric constant or string constant. These constants are often called symbolic constants.

Example: **#define** PI 3.1415

ASSIGNING VALUES TO VARIABLES:

Assignment statement:

This statement is used to assign values to variables. The operator is (=).

Variable _name= value;

Example:

```
Count=0;
Product=1;
Ch='y';
```

Note: More than one assignment statement is allowed in one line.

Ex: I=0;j=0;

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. The general form is:

Datatype vr_name = value;

Examples:

```
int i=0;
char ch='y';
```

Note: The initialization of more than one variable in one statement is possible in **C.** This can be done using multiple assignment operators.

Example: I=j=k=0;

Declaring a variable as Constant:

The value of a variable can be maintained as a constant during the execution of a program. This can be done by using the keyword **const** before the type declaration statement including initial value. The compiler does not allow the defined variable value to be changed.

Example: **const int** array_size = 100;

INPUT AND OUTPUT STATEMENTS:

In **C** there are two types of input statements. They are

- formatted input statements
- un-formatted input statements

Formatted Input statements:

scanf() : This statement is a built-in function in C. It is used to get input values from the keyboard. This requires some format characters to specify the type of data.
The general form is…

scanf("format string", list of addresses of variables);

The **format string** is a specified format code to be read from the keyboard and the list of addresses pof variables is the user defined variable list preceded with **address operator (&)** .These addresses are the address of locations where the data is to be stored. All variables in the scanf() function must be prefixed by **& (address operator)** except certain variables(example, string and pointer variables). Control string and arguments are seperated by commas.

Note:
1. Some variables like strings and pointers must not be preceded with **&** .
2. The values that are sypplied through the keyboard must be separated by either blank(s), tab(s), or newlines(s).
3. The escape sequences cannot be included in the format string.
4. The Format string consists of the conversion character **(%)** , a type specification character .

Data type	Specification character
Integer	d
Float	f
Character	c

Code	Meaning
%c	a single character
%d	a decimal integer
%e	a floating point value
%f	a floating point value
%g	a floating point value
%h	a short integer
%I	a decimal, hexadecimal or octal integer
%o	an octal integer
%s	a string
%u	an unsigned decimal integer
%x	hexadecimal integer
%[…]	a string of word(s)

Example:

Scanf("%d %f %c", &c, &a, &ch);

Width specification: The format can be included in the format string just by specifying the width(w) along with the type specification character.

% w d

Where % conversion character , w is an integer number that specifies the field width , d is the data type character.

Note: Specifying width in the input results in wrong data entry, so this may not be used in most cases.
An input field may be skipped by specifying ***** in the place of field width.

Output statement:

Similar to input, output also has two forms. They are
Formatted output
Unformatted output.

Formatted Output:

The **printf** statement is used to as an output statement that is used to display the required values to the terminal.

The general form is..

Printf("format string", list of variables);

The **format string** can contain
- characters that are simply printed a they are
- conversion specifications that begin with a % sign
- escape sequences that begin with a \ sign

The list of variables should match in number, order and type with that of the format string.
The format specification can also include the width.

The from of using width can be…

% w.p data type

Where *w* is an integer number that specifies the total number of columns for the output value, and *p* is another integer number that specifies the number of digits to the right of the decimal point, (of a real number) or the number of characters to be printed from a string. Both *w* and *p* are optional.

Different forms of output are:

```
        Printf("String");
Ex:     printf("This is the first C class");
        Printf("escape sequence");
Ex:     Printf("\n");
        Printf("control string", list arg);
Ex:     printf("The sum is %d\n", s);
```

Integer data:

Using the formatted statement as follows can print the integer data items:

% wd

where w specifies the minimum field width for the output. If the number is greater than the specified field width, it will be printed in full, overriding the minimum specification. The number is written right-justified in the given field width.

Example: int number;

Printf("%6d", 12345);

_12345

The above statements states that the maximum width of the variable 'number' is 6.

Note: the number can be left-justified by placing a minus sign directly after the % character.

%-wd

Ex: printf("%-6d", 1234);

1234__

Real data:

The format for outputting the float data including width is..

%w.p f

Ex: float avg;
Printf("%7.2f", avg);

In the above statement the avg is of type float having 7 as its total width and from that 2 places are for decimal.

Note: The value when displayed, is rounded to p decimal places and printed right-justified in the field of w columns. Placing a negative sign can do Left justification.

Printing of a Single Character

A single character can be displayed in a desired position using the format

%wc

The character will be displayed *right-justified* in the field of *w* columns. We can make the display *left-justified* by placing a minus sign before the integer *w*.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form

%w.ps

Where *w* specifies the field width for display and *p* instructs that only the first *p* characters of the string are to be displayed. The display is *right-justified.*

Arithmetic Instructions:

These are the instructions where the arithmetic operators are used. To form such type of instructions, the operators should be known thoroughly.

'C'-OPERATORS:

Operators:

All the operators which are used in 'C' are also allowed in cpp various types of C++ operators are as follows.

1. Arithmetic operators
2. Relational Operators
3. Logical operators
4. Conditional Operator
5. Increment and decrement operators
6. Assignment Operators
7. Bitwise operators
8. Special operators

1. Arithmetic operators:

Arithmetic Operators:

These are the basic and common operations in a computer programming language. These are known as binary operators since they require 2 variables to be evaluated. These are

Operator	**meaning**
+	addition
-	subtraction
*	multiplication
/	division
%	modulo(remainder of integer division)

Note: int/int=int
Int/float=float
Float/int=float
Float/float=float

Integer Arithmetic: If both the operands in an arithmetic expression are integers then that expression is called integer arithmetic expression and yields only integer values.

Note: During integer division if both the operands age of same sign the result is towards 'o'. if one of them is negative the direction of truncation is machine dependent.

2. Relational Operators:

The comparison operators are used to compare the values of two variables. These are used in the programme flow. These operators in 'C' produce true (1) or false (0) results. These operators can be grouped as relational and equality operators.

Operator	Meaning	Operator	Meaning
>	greater than	==	equal to
<	less than	!=	not equal to
>=	greater than or equal to		
<=	less than or equal to		

3. Logical Operators:

The various logical operators used are as follows.

Logical AND (&&) : Here the resultant expression can be true only when both the expressions are true.

Logical OR (||) : The resultant expression will be false only when both expressions are false. In all other cases it is true.

Logical Negation (!) : A logical expression can be changed from to false or false to true with negation operation.

4. Conditional Operator:

Syntax:

Exp1 ? Exp2 : exp3

Write a programme to check that the given number is add or even using conditional operator

```
//programme to check the given number is odd or even.
#include<iostream.h>
void main()
{
char *str;
int num;
printf("Enter the given number");
Scanf("%d",num);
str=((num%2==0) ? "even" : "odd");
printf("\n the number is",str);
}
```

5. Increment and decrement operators:

Increment operators: '++' is used as an increment operators and is used increase the value of the variable by '1'. Two types of increment operators are

1. Pre-increment (++ variable)
2. Post-increment (variable ++)

Pre and post increment doesn't differ in the case of just incrementation. But they differ in the case when we are using the value is any operation at the time of incrementation itself.

Pre increment => Increments the value first and does the operation.

Post increment => First does the operation then increases the value

Decrement operators: '- -' is used as decrement operator and decreases the value of variable by '1'. The two decrement operators are

(I) Pre-decrement (- - variable)

(2) Post-decrement (variable --)

Pre and Post decrement doesnot differ in case of just decrement. But they differ in the case when we are using the value in any operation at the time of decrementation itself.

6. Assignment operators:

Operators	Meaning
=	Assign Right hand(RH) value to the left hand(LH) side
+=	value of LH variable will be added to the value of RH and assign back to the variable of LH.
-=	value of RH variable will be subtracted from the value of LH and assign back to the variable of LH.
*=	value of LH variable will be multiplied by the value of RH and assign back to the variable of LH.
/=	value of LH variable will be divided by the value of RH and assign back to the variable of LH.
%=	the remainder will be stored back to the LH after integer.

7. Bitwise Operators:

The operators which work as bit level are known as Bitwise operators.

Bitwise AND: (&): This operator returns a true value if both are true else it returns 'o'.

a=0000 0000 0000 0101

b=0000 0000 0000 0111

a & b = 0000 0000 0000 0101

Bitwise OR (|): This operator returns the value '!' if atleast one of the value is 1 else it returns zero.

a/b = 0000 0000 0000 0111

Bitwise Exclusive OR (^): This operator returns value 1 if only one of the value is 1, else it returns 0.

a^b = 0000 0000 0000 0010

Bitwise complement (~) or Tild: It returns the value '1' for '0' and '0' for '1'.

~a = 1111 1111 1111 1010

Bitwise right shift operator (>>): In this operator one value in the right side is vanishes i.e. a very digit shifts one bit right and left gap will be filled with zeros.

a=0000 0000 0000 0101
>>a=0000 0000 0000 0010

Bitwise left shift operator (<<): In this operator one value in the left side is vanishes i.e every digit shifts one bit left and right gap will be filled with zeros.

b=0000 0000 0000 0111
<<b=0000 0000 0000 1110

Special Operators:

Some special operators who are used in cpp are as follows.

Comma operator (,) :

- Comma is used as a separator in the variable declaration.
- Used as a separator in the list of variables in input and out statements.
- Used to link the related expressions together.
  A comma-linked list of expressions are evaluated left to right and the value of rightmost expression is the value of combined expression.

Pointer Operator : (*) This is used to declare pointer variable and also used to get the value at the address. (pointer is a variable which is used to store the address of another operator).

Size of Operator: The size of operator is a compile time operator, and when used with an operant, it returns the number of bytes the operand occupies.

Note: This is normally used to determine the size of arrays and structures.

2. It is also used to allocate memory space dynamically to a variable during execution of a program.

Type conversion:

In mixed mode expressions the operands are converted to its higher data type before the compilation takes place, to maintain compatibility b/w the datatypes. This can be done in two ways. They are

1. Implicit type conversion
2. Explicit type conversion

1. Implicit type conversion: The compiler performs type conversion of data items when an expression of data items of different types this is called implicit or automatic type conversion.

Rule table for implicit conversion:

operand 1	Operand 2	Result
Char	int	int
Int	long	long
Int	float	float
Int	double	double
Int	unsigned	Unsigned
Long	double	Double
Double	float	double

Note: In a mixed mode expression the resultant value is automatically converted to the type of the variable in LHS.

Example

```
float  j;
int i,k;
J=9,k=2;
I=j/k;
```

Here, the value of variable is converted to float and floating point division takes place., then the result is 9.0/2.0 = 4.5. But, the LHS variable i is of type integer. So, result the decimal part is truncated 4 is assigned to i.

Explicit type conversion: The implicit conversion sometimes leads to errors in the program. So, the use of explicit type conversion is advised in mixed mode expressions.

Type cast operator:

The explicit type conversion can be implemented by using type cast operator.

Syntax: (datatype) expression;

Or

(datatype) variable name;

This syntax is followed in 'c'. And is also allowed in cpp

In cpp, there is another syntax for type conversion, which is specific to cpp.

Syntax: datatype (expression);

Or

datatype (variable name);

Eg: float (i);
float (i/j);
int (j);

Type casting can also be used to convert higher data type to lower data type.

Control Statements

In C language there are two types of control statements are there. They are

- Decision Making and Branching
- Decision Making and Looping

Decision Making and Branching

In the situations like where the order of execution of statements has to be changed based on certain conditions. This can be done in C using the decision-making statements such as

- if statement
- switch statement
- conditional operator statement
- goto statement

Note: **if** statement is called conditional branching statement and **goto** statement is called unconditional branching.

The *if Statement:*

This statement is a decision-making statement and is used to control the flow of execution of statements.

There are different forms of **if** statements. They are

- Simple **if** statement
- **If…else** statement
- Nested **if…else** statement
- **Else…if** ladder.

Simple if statement:

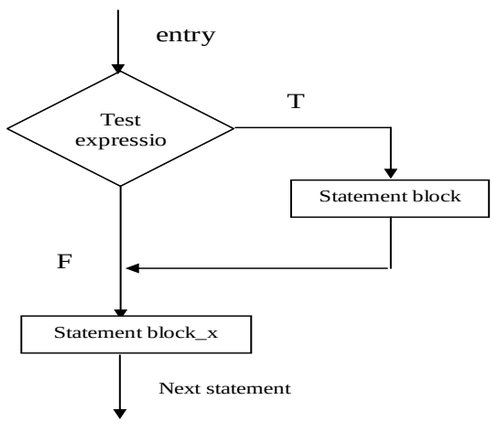
The general form of this statement is :

```
If(condition)
{
  Statement-block;
}
statement-x;
```

Here if the condition is evaluated first and if it is true the statement block is executed and proceeds to statement-x, (either single or block of statements) otherwise (if the condition is false) the block is skipped and the following statement i.e., statement-x is executed.

Example:

```
If(medium==’Telugu’)
   Marks=marks+5;
Printf(“%f”, marks);
```



Write a program to find the ratio of a+b to c-d

```
/*Program to find the ratio of a+b to c-d*/
#include<stdio.h>
main()
{
        int a,b,c,d;
        float ratio;
      printf(“Enter the integer values to a,b,c,d:\n”);
        scanf(“%d%d%d%d”,&a,&b,&c,&d);
        if(c-d!=0)
        {
        ratio=(float)(a+b)/(c-d)
        printf(“%.2f”, ratio);
        }
}
```

The if…else Statement:

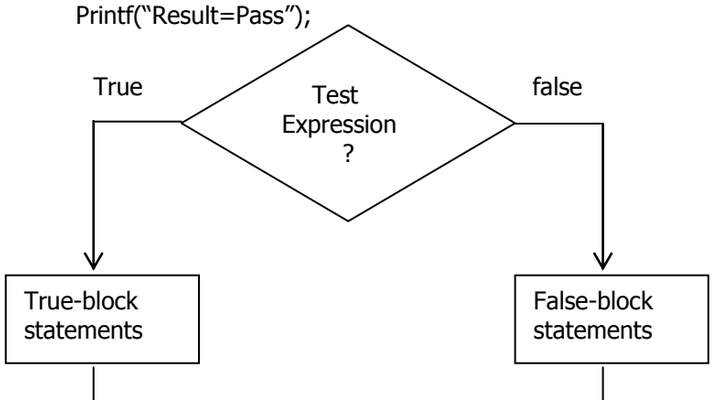
This statement is an extension to the simple if. The general form is..

```
If(condition)
{
  true-block statements
}
else
{
  false-block statements
}
statement-x;
```

Here, first the condition is evaluated, if it is true the true-block is executed, if the condition is false then false-block is executed. In both the cases the control is transferred subsequently to statement-x.

Example

```
If(marks>=35)
        Printf(“Result=Pass”);
Else
        Printf(“Result=Pass”);
```



Nested if…else:

When more than one decision is involved, more than one if statement has to be used. Then nested if statement is used. The general form of this statement is…

```
If(condition 1)
{
        if(condition 2)
        {
          statement-1;
        }
        else
        {
          statement-2;
        }
}
else
{
        statement-3;
}       statement-x;
```

Here, if condition-1 is true then condition-2 is tested, if it is true then statement-1 is executed, otherwise if condition-2 is false, statement-2 is executed. If condition-1 itself is false, then statement 3 is executed. In all the cases statement-x is executed.

Example:

```
If(a>b)
        {
          if(a>c)
            Printf("%f\n",a);
          else
            Printf("%f\n",c);
        }
        else
        {
          if(c>b)
            Printf("%f\n",c);
          else
            Printf("%f\n",b);
        }
```

The Else…if ladder:

This is another way of putting **if**s together when multipath decisions are involved. A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if**. The general form is…

```
If(condition-1)
  Statement-1;
        Else if(condition-2)
          Statement-2;
                Else if(condition-3)
                  Statement-3;
                                Else if(condition-4)
                                  Statement-n;
                                        Else
                                          Default-statement;
Statement-x;
```

Here, whenever a condition is true, the statement associated is executed and the control is transferred to statement-x, and if it is false another condition is tested and so on…

Example:

```
If(marks>=80)
        Printf("Result=Distinction");
Else If(marks>=60)
        Printf("Result=First");
Else If(marks>=50)
        Printf("Result=Second");
Else If(marks>=40)
        Printf("Result=Third");
Else
        Printf("Result=Fail");
```

Switch statement:

The program will be complicated when multiple conditions are involved and number of alternatives are increased. If constructs reduces the readability. C provides a multi-way decision statement i.e., **switch.**

The switch statement tests the value of a given variable against a list of case values and when a match is found, a block of statements associated with the case is executed. The general format is…

```
switch(expression)
{
    case value-1:
        block-1;
        break;
    case value-2:
        block-2;
        break;
    ..............
    defalut:
        default-block;
    }
statement-x;
```

Here, the expression is an integer expression and it produces an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others.

First, the integer expression following the keyword **switch** is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statement. When a match is found, the program executes the statements following the Case, and all subsequent case and default statements as well. If no match is found with any of the case statements, only the statements following the default are executed.

Example:

```
Main()
{
     char ccode;
    Ccode=getch();
    Switch(ccode)
   {
      case 'r' :
                printf("Red\n");
                break;
      case 'b' :
                printf("Blue\n");
                break;
      case 'g' :
                printf("Greeen\n");
                break;

      default:
                printf("Wrong code\n");
    }
}
```

Decision Making and Looping:

When there is a need of repeating set of statements more than once, looping structures should be used. This repetition is either a specified number of times or until a particular condition is being satisfied.

There are three types of control structures in 'C'. They are:

- Using a **for** statement
- Using a **while** statement
- Using a **do-while** statement

while statement:

This form is used to form a loop. This is the simplest form of the all loop structures.

The general format is…

```
While (condition)
{
  body of the loop
}
```

This is an entry-controlled loop statement. First the condition is evaluated and if the condition is true, then the body of the loop is executed. After the execution of the body of the loop, condition is once again evaluated and if it is true, the body is executed once again. This process continues until the test-condition finally becomes false. The control passes to the statement following the loop.

Note: The braces are required only the body of the loop contains more than one statement.

Ex:

```
I=1;s=0;
While(I<=n)
{
  s=s+I;
  I=I+1;
}
```

The do statement:

In the case of while statement the body of the loop will not be executed even once if the condition is false in the beginning. But in some cases, the body needs to be executed at least once. In such situations an exit-controlled loop can be used. In 'C' do statement is of such type.

The general syntax is….

```
do
{
body of the loop
}
while (condition);
next statement;
```

Here, first the body is executed and at the end the condition is tested. If it is true, the loop continues to execute and if the condition is false, the loop terminated and the control goes to the statement following the loop.

Ex:
```
do
{
    ch=getch();
    line[I] = ch;
    I++;
}
while(ch!='#');
```

The for statement:

This is another entry-controlled loop. The for loop allows us to specify three things about a loop in a single line. They are:

- setting a loop counter to an initial value
- testing the loop counter to determine whether its value has reached the number of repetitions
- Increasing the value of loop counter each time the program segment within the loop has been executed.

The general format is…

```
for(initialization; condition; increment)
{
        body of the loop
}
```

The execution of the **for** statement is as follows:

- Initialization of the control variable is done first, using assignment statement such as I=1 and c=0. The variables I and c are known as loop-controlled variables.
- The value of the control variable is tested using the condition. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as I=I+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the loop is executed again till the condition fails.

Ex:
```
For(j=0;j<=10;j++)
  Printf("%d/n", j);
```

Forms of for loop:

- More than one variable can be initialized at a time in the initialization part.

Ex:
```
For(I=0, j=1;I<10;I++)
```

- Like initialization section, the increment section may also have more than one part.

Ex: for(I=0.j=10;I<10;I++,j--)

- The condition may have any compound relation and the testing need not be limited only to the loop control variable.

Ex:
```
for(I=1,s=0;I<10 && s<20; I++)
 {
   s=s+I;
 }
```

- The important aspect of for loop is that one or more sections can be omitted, if necessary.

Ex:
```
for(;I != 20;)
 {
   I=I+2;
 }
```

Break statement (jumping out of loop):

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The syntax is….

break;

Ex:
```
if (x<0)
break;
```

Continue statement:
This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

The syntax is……

continue;

Ex:
```
if(x<=0)
{
  printf("invalid input");
  continue;
}
```

Arrays

"An array is a group of similar type of data items stored in consecutive memory locations that share a common name".

(or)

"A group of homogeneous data items stored under a single name is called an array".

(or)

"An array is a fixed-size sequenced collection of elements of the same data type".

Examples:

- List of employees in an organization.
- List of products and their cost sold by a store.
- List of customers and their telephone numbers.

The single values in the array are called as elements. The number of elements in an array can be represented as an index with in the square brackets immediately after the array name.
Note: Individual array elements are identified by an integer index(subscript). In C the index (subscript) begins with zero and is always written inside square brackets.

Declaration of Array:

Similar to other variable arrays must be declared before it is used in a programme. The array declaration is defining the type of the array, Name of the array, the number of elements in the array.
Syntax is

datatype arrayname[size];

Here data type is the type of the data elements in the array and the arrayname indicates the name, size number of elements that are stored inside the array.
Example:

```
int age[20];
float height[20];
char name[10];
```

Types of Arrays:

There are different types of arrays depending on the number of subscripts for representing the element of the array. They are

1. One dimensional array
2. Two dimensional array
3. Multi dimensional array

- A list of items that can be given one variable name using only one subscript is called a single subscripted variable or one dimensional array
  Ex: int N[15];
- A two dimensional array is one which in represented by using two subscripts and can be used to store a table of values.
- An array, which has three or more dimensions is known as three dimensional or multi dimensional array.

ONE-DIMENSIONAL ARRAYS:

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array.
Declaration: Similar to the normal variables array variables can be declared. The general form is…

datatype var_name[size];

Here, type specifies the data type of the elements that the array contains, size indicates the maximum number of elements that the array can have.
Ex:

int results[20];

Initialization:

Same as ordinary variables, array variables can be initialized when they are declared.

Syntax:

type array_name[size] = {list of values};

Ex:

int num[3]={4,5,6};

char name[4]={'R','A','J','U'};

Note: It is not possible initialize only specified elements of the array.

The array size can be omitted which initializing the array. In this case the size is known to compiler by considering the initial values given to the array.

float marks[5] = { 46.5, 36.5, 40 }

Note: If the number of initial values are less than the size of the array then the last elements are filled with zeros.

- The order of the initial values should be matched with the order of the elements

Processing of Arrays:

The arrays can be processed by using the for loop. Processing of arrays include reading, writing and manipulating of arrays.

```
Ex:  int x[20];
     for(i=0;i<20;i++)
          Scanf("%d", &x[i]);
     for(i=0;i<20;i++)
          Printf("%d",x[i]);
```

TWO-DIMENSIONAL ARRAYS:

Arrays can have more dimensions. If a table of values have to be stored in the then the two dimensional array is required.

Declaration: The two-dimensional array can be declared as..

datatype array_name [row_size][column_size];

Same as single dimensional array, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

Initialization:

int matrix[3][3]={1,2,3,4,5,6,7,8,9}

or

{ {1,2,3},{4,5,6},{7,8,9}}

Note: C allows arrays of three more than three dimensions. This can be declared as

int results_3d[20][5][3];

Each index has its own set of square brackets.

Processing:

The two dimensional arrays can also be processed using nested for loops.

```
Ex:  for(i=0;i<m;i++)
          for(j=0;j<n;j++)
            Scanf("%d",&m[i][j]);
```

Multi-Dimensional Arrays

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

datatype array_name[s1][s2][s3]….[sm];

Where s; is the size of the ith dimension. Some examples are:

int survey[3] [5] [3];

float table [5] [4] [3];

Survey is a three-dimensional array declared to contain 180 integer type elements. Similarly table is a four-dimensional array containing 300 elements of floating-point type.

The array survey may represent a survey data of rainfall during the last three years from January to December in five cities.

Strings

A string is an array of characters.

or

Any group of characters(except double quote sign) defined between double quotation marks is a constant string.

Declaring string variables:

A string variable is any valid C variable name and is always declared as an array. The general form of declaration of a string variable is

char string_name[size];

the size determines the number of characters in the string_name.

Ex: char city[10];

char name[20];

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

Initializing string variables

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

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

The reason the city had to be 9 element long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator.

Note: when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized.

Ex: static char string[]={'G','O','O','D','\0'};

The above statement defines the array string as a five element array.

Note: The word static may be omitted when using ANSI compilers.

Reading a String from terminal:

The familiar input function **scanf** can be used with %s format specification to read in a string of characters.

Ex: char address[15];
Scanf("%s",address);

The problem with the scanf function is that it terminates its input on the first white space it finds. Therefore, if the following line of text is typed in at the terminal,

NEW YORK

Then only the string "NEW" will be read into the array **address,** since the blank space after the word **NEW** will terminate the string.

Note: In the case of charcter arrys, the ampersand (&) is not required before the variable name.

Writing Strings to Screens:

We have used extensively the printf function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character.

Ex: printf("%s",name);

We can also specify the precision with which the array is displayed. For instance, the specification

%10.4s

indicates that the first four characters are to be printed in a field width of 10 columns.

However, if we include the minus sign (e.g.,%-10.4s), the string will be printed left justified.

Arithmetic operations on characters:

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system.

(a) To write a character in its integer representation:

Ex: if the machine uses the ASCII representation. Then

```
x='a';
printf("%d",x);
```

will display the number 97 on the screen

(b) To perform arithmetic operations:

It is possible to perform arithmetic oprations on the character constants and variables

Ex: **x='z'-1;**

Is a valid statement. In ASCII, the value of 'z' is 122 and therefore, the statement will assign the value 121 to the variable x.

(c) In relational expressions:

We may also use character constants in relational expressions. For example, the expression

ch>='A' && ch<='z'

Would test whether the character contained in the variable **ch** is an upper case letter.

(d) Converting string to digits to interger

The C library supports a function that converts a string of digits into their integer values. The function takes the form

X=atoi(string)

Here x is an integer variable and sting is a character array contaiing a string of digits. Ex:

```
number="1998";
Year=atoi(number);
```

Number is a string variable which is assigned the string constant "1998". The function **atoi** converts the string "1998" to its numeric equivalent 1998.

CHARACTER FUNCTIONS:

Sometimes, it may be required to analyze a character type such as alphabetic, alphanumeric, numeric[digit], control characters etc. These facilities are declared in the header file **<ctype.h>**. Basically, there are two types of handling features available in the **<ctype.h>** such as classification and conversion. Every character a value of type **int** i.e., non-zero if the argument is in the specified class, and zeros if not. Every character conversion facility has a name beginning with "**to**" as return a value of type int representing a character (or) EOF.

isalnum():

It is used to check weather a given character is an alphanumeric character (or) not. The function writtens non-zero value if the character belongs to an alphanumeric otherwise the returned valued is zero.

Alhanumeric characters:

A, ---------- Z; a.----------z; 0----------9:

Example: 1.isalnum(a) ------→ 1 (true)
2.isalnum(-) -------→ 0 (false)

isalpha( )

it is used to check weather a given character belongs to an alphabetic character, if it is true, it returns non-zero otherwise the returned value is zero.

Alphabetic characters:
A,----------Z; a,----------z

Example: isalalpha(a) ------→ 1 (true)
isalpha(-) -----→ 0 (false)

isdigit( )

it is used to check weather a given character belongs to a decimal digit, if it is true, it returns a non-zero value other wise the return value is zero.

The digits: 0,1,2,----------9

Example: isdigit(9) -----→ 1 (true)
isdigit(b) -----→ 0 (false)

islower( ):

It is used to check weather a given character belongs to a lower case alphabetic character or not. If it is true it returns non-zero otherwise the return value is zero, the lower case alphabets are,
a,b,----------,z

Example: islower(a) -------→ 1(true)
islower(b) -------→ 0(false)

isupper( ):

It is used to check weather a given character belongs to a upper-case alphabetic character or not. If it is true ,it returns non-zero otherwise the returned value is zero. The upper case characters are:
A,B,----------Z

Example: isupper(Z) -------→ 1 (true)
isupper(y) -------→ 0(false)

ispuct( ):

It is used to check weather a given character belong to a punctuation character (or) not. If it is true the returned value is non-zero otherwise it returns zero. The punctuation characters are
., . : ; -------

Example: ispuct(--) ----------→ 1(true)
ispuct(A) ----------→ 0(false)

isspace( ):

It is used to check weather a given character belongs to a white space character or not. If it is true, it returns non-zero value otherwise the returned value is zero.

The white space character are
horizontal tab, vertical tab, newline, formfeed, carriage return & space etc.

Conversions:

tolower( ):

it is used to convert a given upper case letter into a corresponding lower case letter. If a given character is not an upper case letter then it returns the character unchanged.

Example: tolower(T) 1 t
tolower(6) 0 6

toupper( ):

it is used to convert a given lower case letter into a corresponding uppercase letter. If a given character is not a lower case letter then it returns the character unchanged.

Example: toupper(t) 1 T
toupper(A) 0 A
toupper(5) 0 5

Write a program to count the number of characters, vowels, consonants, digits, tabs, spaces and words in a given string.

/*Program to count number of characters, vowels, consonants, digits, tabs, spaces and words*/

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
main( )
{
        char line[51], ch;
        int c, vowels=0, consonants=0, char_count=0;
        int digits=0, spaces=0, words=0, punct=0;
        c=0;
        clrscr();
        printf("Enter the text < press return at end>:\n");
        do
        {
        ch=getchar( );
        line[c]=tolower(c);
/*Finding vowels & consonants*/
        if(isalpha(ch)>0)
```

```
        {
          switch(tolower(ch))
          {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                        vowels++;
                        break;

             default:
                        consonants++;
                        break;
           }
        }
/*Finding of digits*/
        if(isdigit(ch) >0)
        {
                digit++;
        }
/*Finding spaces*/
         if(isspace(ch)>0)
        {
                space++;
        }
         if((line(c)==' ')||(line(c)=='.'))
        {
                words++;
        }
/*Finding special characters*/
         if(ispunct(ch>0))
        {
                punct++;
        }
                c=c+1;
      } /*closing of do*/
      while(ch!='\n');
      c=(c-1);
      line(c)='\0';
      printf("\n");
      printf("no of characters:%d\n", c);
      printf("no of consonants:%d\n", consonants);
      printf("no of vowels:%d\n", vowels);
      printf("no of digits:%d\n", digits);
      printf("no of spaces:%d\n", spaces);
      printf("no of words:%d\n", words++);
      printf("no of punctuation:%d\n", punct);
} /*closing of main*/
```

String Handling Functions

Strcat( ): This function joins two strings together. It takes the following form.

strcat(str1,str2)

(OR)

strcat(str1,str2,n)

str1,str2, are character arrays. When the function strcat( ) is executed, str2 is appended to str1. It does so by removing the null character at the end of the str1 and placing str2 from there. The string at str2 remains unchanged. Where **'n'** is optional. To specify how many characters of str2 is to added str1, then only we specify the **'n'** value.

Example: strcat(str1,str2) (or) strcat(str1,str2,n)

Str1 =

S	A	I	\0			

Str2 =

R	A	M	\0			

Strcat(str1,str2);

Result:

Str1

S	A	I	R	A	M	\0

R	A	M	\0			

Str2

Strcmp( ): The string compare function is used to lexo graphical compare the contents of the null terminated str1 with the contents of null terminated str2. It returns value of type **int** of any one of the following.

Str1==str2 returns 0
Str1<str2 returns -1
Str1>str2 returns 1

Example: 1. strcmp("reta","RETA");
Returns 1 (str1>str2)
2. Strcmp("RETA","reta");
Returns -1(str1<str2)

Strcpy( ): The strcpy( ) is used to copy the contents of the str2 to the str1. The entire contents of str2 are copied, plus the terminating null, even if str2 is larger than str1.

Example:

char str1[5];
Str2[5]="roja";
strcpy(str1,str2);

Result:
Str2 is assigned to str1.so the str1 has the string "roja"

Strlen( ): The strlen( ) is used to find the number of characters in a given string preceding the terminating. The length of an empty string is zero.

Str1="ROJA"
Strlen(str1);
it returns an integer number 4;

Two-dimensional array:

This can be used to store a list of data items. The general syntax is

char string name[l-size][e-size] ;

Where l-size refers the number of elements to be stored in the array e-size refers the each element length. Array elements may be initialized.

Example: char city[6][20]={"BOMBAY","HYDERABAD","DELHI","BOMBAY","PUNE","DELHI"};

Where the list size is optional.

Example: char city[][20]= {"BOMBAY","HYDERABAD","DELHI","BOMBAY","PUNE","DELHI"};

/*program to sort list of names*/

```
#include<stdio.h>
#include<string.h>
#define NAMES     10
#define MAX_CHARS    25
main( )
{
        char name[NAMES][MAX_CHARS];
        char temp[MAX_CHARS];
        int n,i,j;
        printf("How many names are there:");
        scanf("%d", &n);
/*storing of names in alphabetical order*/
        for(i=0;i<n-1;I++)
        {
          for(j=i+1;j<n;j++)
          {
            if(strcmpi(name[i],name[j])>0)
            {
              strcpy(temp, name[i]);
              strcpy[name[i],name[j]);
              strcpy[name[j],temp);
            }
          }
        }
        printf("\n sorted list:\n");
        for(i=0;i<n;i++)
        {
        printf("name[%d]:%s", i+1, name[i]);
        }
}
```

FUNCTIONS

Def: "A function is a self-contained block of code that performs a particular task".

C functions can be classified into two categories, namely,

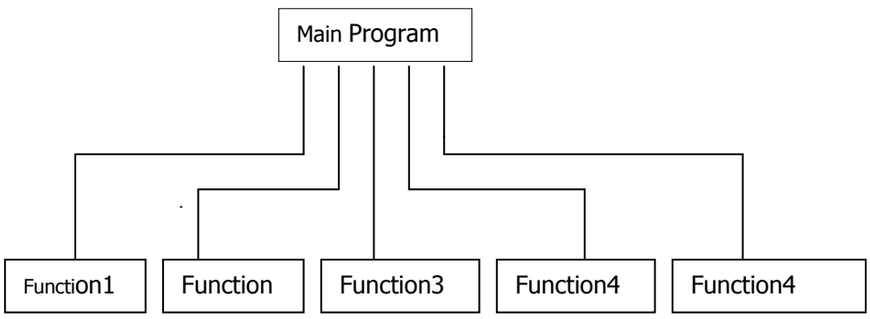
- **Library functions** or **built-in functions**
- **User-defined functions**.

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

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

Advantages of functions:

- Simple to write .
- Easy to read, write and debug.
- It facilitates top-down modular programming. In this style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
- Using functions at appropriate places can reduce the length of a source program.
- A function may be called any number of times in any place with different parameters in a program.
- Many other programs may use a function. This means that a C programmer can build on what others have already done, instead of starting over, from scratch.
  The relation of functions with a main program can be shown as follows .



User-defined functions:

A user-defined function mainly contains three parts. They are

- Function definition: An independent program module that is specially written to implement the requirements of the function.
- Function call: Used to invoke it at a required place in the program.
- Function declaration: The declaration of the function in the calling program.

Definition of Functions:

A function definition contains the following elements.

- Function name
- Function type
- List of parameters
- Local variable declarations
- Function statements
- A return statement

In the above three six elements, first three are grouped and **Function header** and the other three elements are called as **Function body**.

The general format:

```
Function_type function_name(argument list)  -----→ Function Header
 Argument declaration;
 {
   local variable declaration;
   executable statements;
     ……….
   return statement;
 }
```

New ANSI C compilers allows the declaration of arguments with in the parenthesis itself.
i.e.,

```
function_name(datatype arg-1, datatype arg2,….)
{
        body of the function;
}
```

In the above syntax the argument list and the declaration of local variables and 'return' statement are optional.

- The argument list is not required if the function has no arguments.
- The local variable declaration is not necessary if the function has no new variables.
- The return statement is optional in the case where the function does not return anything.

Note: The function is called as "called program" and the main program, which includes this function, is known as "calling program".

Function Header: It consists of three parts. They are function type, function name and the formal parameter list.

- The **function type** specifies the type of value that the function is expected to return to the program calling the function.
- The default return type is int. It is assumed, if any return type is not specified.
- If the function is not returning anything, then we must specify the return type as **void**.
- The **function name** is any valid C identifier.

Formal Parameter List (or) Argument list:

- The parameter list declares the variable that will receive the data sent by the calling program. They work as input data to the function. These are also used to send values to the calling programs. The parameters are also known as arguments.
- The parameter list contains declaration of variables separated by commas and surrounded by parentheses.

```
Ex:  int sum(int x, int y)          float mul(float a, float b)
     {                               {
          ……..                            ……
      }                               }
```

- This function neither receives any input values not returns back any value, then it can be specified with void.

Function Body:

- The function body is enclosed in braces and contains three parts. They are,
- Local variable declaration
- Executable statements
- A return statement. If the function does not return any value, the return type should be specified as **void.**

RETURN VALUES AND THEIR TYPES:

A function may or may not return a value. If it is, then it is done with a **return** statement. But it can return only one value.

Syntax:

return;
Or
return(expresion);

the first statement is used when the function does not return any value.

Ex: if(error)
return;

the second statement returns with an expression that returns the value of an expression.

Ex:

```
Mul(int x, int y)
{
  int p;
   p=x*y;              or    return(x*y);
   return(p);
 }
```

Function Calls:

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

Ex: mul(m,n);

Note:

- A function, which returns a value, can be used in expressions like any other variables.
- A function cannot appear on the left-hand side of the assignment operator.
- A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function.

Function Declaration & Prototype(Handling non-integer functions):

Any C function by default returns an integer value. If the return type is not specified then also it is considered. If we require a function to return other than an integer value then the function prototype is required.

The function declaration is also known as function prototype. All the functions in a C program must be declared, before they are invoked. A function declaration consists of four parts. They are

- Function type or return type
- Function name
- Parameter list
- Terminating semicolon

Syntax:

Function return type function name(parameter list);

The called function must be declared at the beginning of the body of the calling function like any other variable. This tells the calling function, the type of the data and its return value.

Categories or Functions:

Based on whether arguments are present or not and whether a value is returned or not, the functions are categorized as follows.

- Function with no arguments and no return values.
- Functions with arguments and no return values.
- Functions with arguments and one return value.

FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUES:

The functions, which do not receive any values from the calling program will have no arguments and the calling function which does not receive any value from the called function i.e., which have no return values are come into this category. When a function has no argument it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. So, there is no data transfer between the calling function and the called function.

Ex: /*program to illustrate NO ARGUMENTS AND NO RETURN VALUES */

```
#include<stdio.h>
main()
```

```
        {
            pline();
            printf("WELCOME");
            pline();
        }
        pline()
        {
            int i;
            for(i=0;i<30;i++)
                printf("%c", '@');
            printf("\n");       }
```

output:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
WELCOME
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUES:

Some functions can receive data from the calling program but cannot send any data to its calling program. Such functions come into this category. This approach sees function can check for validity of data, if necessary, before it is handed over to the called function. The nature of data communication between the calling function and called function with arguments but no return value is one way data communication.

Ex:

```
/*program to illustrate  WITH ARGUMENTS BUT NO RETURN VALUES      */
      #include<stdio.h>
      main()
        {
            pline(40,'@');
            printf("\t\tWELCOME");
            pline(50,'&');
        }
        void pline(int n, char ch)
        {
            int i;
            for(i=0;i<n;i++)
                printf("%c", ch);
            printf("\n");
        }
```

output:
@@
WELCOME
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

FUNCTIONS WITH ARGUMENTS AND WITH RETURN VALUES

The functions, which receive values from the calling function and which returns values to calling function will come into this category. These functions will have two way data communication.

Here is a function, which raises a **double** to the power of an unsigned, and returns the result.

```
double power(double val, unsigned pow)
{     double ret_val = 1.0;
      unsigned i;
      for(i = 0; i < pow; i++)
            ret_val *= val;
      return(ret_val);
}
```

The function follows a simple algorithm, multiplying the value by itself pow times. A for loop is used to control the number of multiplications, and variable ret_val stores the value to be returned. Careful programming has ensured that the boundary condition is correct too. ie .

Let us examine the details of this function.

Double power(double val, unsigned pow) This line begins the function definition. It tells us the type of the return value, the name of the function, and a list of arguments used by the function. The arguments and their types are enclosed in brackets, each pair separated by commas.

The body of the function is bounded by a set of curly brackets. Any variables declared here will be treated as local unless specifically declared as static or extern types.

return(ret_val);

On reaching a return statement, control of the program returns to the calling function. The bracketed value is the value, which is returned from the function. If the final closing curly bracket is reached before any return value, then the function will return automatically, any return value will then be meaningless.

The above function can be called as **result = power(val, pow)**;

This calls the function power assigning the return value to variable result.

Actual arguments & Formal arguments:

The parameters that are passed from a calling program to the function are called **Actual arguments.**

```
Ex:  main()
        {
         int x,y;
        …….
```

```
…….
Output(x,y);  /* x and y are actual arguments */
}
```

The **Formal arguments** are those parameters that are present in a function definition and it may also be called as dummy arguments.

```
Ex:  main()
{
 int x,y;
……..
…….
Output(x,y);
}
output(int a, int b)        /* a and b are formal arguments */
{
……..
……..
}
```

Scope of Function Variables:

Local Variables:

Variables declare within the function are called local variables. The names of local variables have meaning only within the function in which they are declared. These are referred only to the particular part of the function for block. The same variable name can be given to different parts of a function for a block.

Local variables are declared within a function. They are created newly each time the function is called, and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables

```
Ex: fun1(int I, int j)
      {
         int p,q;
         q=21;
         p=45;
     }
  fun2(int x, int y)
   {
         int p,q;
         p=36;
         q=57;
   }
```

Global Variables:

The variables that can be accessed by any function in the program are called global variables. As these variables are not declared with in a specified function, they are accessible to all functions in the program. Therefore global variable declaration is outside of any function. This indicates global nature. That is it does not belong to any function.

Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

/*program to find the sum*/

```
#include<stdio.h>
int x,y=5;
main()
{
   x=15;
   f1();
   getch();
 }
f1()
{
  int s;
  s=x+y;
  printf("\n Sum of %d and %d is %d \n",x,y,s);
}
```

Modifying Function Arguments

Some functions work by modifying the values of their arguments. This may be done to pass more than one value back to the calling routine, or because the return value is already being used in some way. C requires special arrangements for arguments whose values will be changed. You can treat the arguments of a function as variables, however direct manipulation of these arguments won't change the values of the arguments in the calling function. The value passed to the function is a copy of the calling value. This value is stored like a local variable, it disappears on return from the function.

There is a way to change the values of variables declared outside the function. Passing the addresses of variables to the function does it. These addresses, or pointers, behave a bit like integer types, except that only a limited number of arithmetic operators can be applied to them. They are declared differently to normal types, and we are rarely interested in the value of a pointer. It is what lies at the address which the pointer references which interests us.

To get back to our original function, we pass it the address of a variable whose value we wish to change. The function must now be written to use the value at that address (or at the end of the pointer). On return from the function, the desired value will have changed. We manipulate the actual value using a copy of the pointer

Handling of Non-Integer function (Function declaration and prototype):

Any 'C' function by default returns an 'int' value more specifically whenever a call is made to a function, the compiler assumes that this function would return a value of the type int.

If we desire that a function should return a value other than an int it is necessary to explicit type-specifier, corresponding to the data type required must be mentioned in the function header. The **general form of this function definition is**

Type-specifier function-name(argument list)
Argument declaration;
{
function statement;
}

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

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data that the function is actually returning.

Example: perform multiplication and division operations on two variables x & y or type that.

```
main( )
{
   float a,b,mul( ),div( );
   a=12.345;
   b=9.82;
   printf("%f\n", mul(a,b));
   printf("%f\n",div(a,b));
}
float mul(x,y)
float x,y;
{
  float p;
  p=x*y;
  return(p);
}
float div(x,y)
float x,y;
{
   return(x/y);
}
```

Example: Square of a number

```
main( )
{
   float square( );
   float a,b;
   printf("Enter a number");
   scanf("%f", &a);
   b=square(a);
   printf("%f",b);
}
float square(x)
float x;
{
  float y;
  y=x*x;
  return(y);
}
```

Nesting of functions: 'C' permits nesting of functions freely. Main can call function1, which calls function2 which calls functions and so on. There is in principle no limit as to how deeply functions can be nested.

Recursive Functions

Def: Recursion

Recursion is a process by which a function repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

(or)

A recursive function is one that calls itself. We shall provide some examples to illustrate recursive functions. Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non-recursive version of this earlier.

```
/* Fibonacci value of a number */
int fib(int num)
```

```
{       switch(num) {
        case 0:
                return(0);
                break;
        case 1:
                return(1);
                break;
        default:  /* Including recursive calls */
                return(fib(num - 1) + fib(num - 2));
                break;
        }
}
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{
        if(pow == 0)  /* pow(x, 0) returns 1 */
                return(1.0);
        else
                return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly, otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions, which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the fibonacci function of a moderate size number.

If such a function is to be called many times, it is likely to have an adverse effect on program performance. Don't be frightened by the apparent complexity of recursion. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

STORAGE CLASSES:

Based on the scope and longevity of the variables, C variables are classified as four storage classes. They are

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

The **scope** of variable determines over what part(s) of the program a variable is actually available for use (active). **Longevity** refers to the period during which a variable retains a given value during execution of a program (alive). So longevity has a direct effect on the utility of a given variable.

The variables may also be broadly categorized, depending on the place of their declaration, *as internal* (local) or *external* (global). Internal variables *we* those which are declared within a particular function, while external variables are declared outside of any function. It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are *created* when the function is called and *destroyed* automatically when the function is exited, hence the name **automatic**. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as *local or internal* variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable number in the example below is automatic.

```
int number;
```

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

```
auto int number;
```

There are two consequences of the scope and longevity of auto variables worth remembering. First, any variable local to main will normally live throughout the whole program, although it is *active* only in main. Secondly, during recursion, the nested variables are unique auto variables, a situation similar to function-nested auto variables with identical names.

Automatic variables can also be defined within a set of braces known as "blocks". They are meaningful only inside the blocks where they are defined. Consider the example below

```
main()
{
  int n, a, b;
  ..............
    if(n <= 1 00)
   {   int n, sum;
     } /*sum not valid here */
}
```

The variables n, a and b defined in main have scope from the beginning to the end of main. However, the variable n defined in the main cannot enter into the block of scope level 2 cause the scope level 2 contains

another variable named n. The second n (which takes precedence over the first n) is available only inside the scope level 2 and no longer available moment control leaves the if block.

Storage: memory

Default initial value: Unpredictable value, which is often called the garbage value.

Scope: local to the block in which the variable is defined.

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

External Variables

Variables that are both ***alive*** *and* **active** throughout the entire program are known as ***external*** **variables**. They are also known as *global* variables. Unlike local variables, global variables be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer number and float length might appear as: lnt number;

```
float length = 7.5;
main()
{
}
```

When a function is defined in one file and accessed in another, the later file must include a function *declaration.* The declaration identifies the function as an external function whose definition appears elsewhere. We usually place such declarations at the beginning of the file, before all functions. Although all functions are assumed to be external, it would be a good practice to explicitly declare such functions with the storage class extern.

The variable number and length are available for use in all the three functions. In case a local variable and a global variable have the same name, the local variable will have precedence over the global variable have the same name, the local variable will have precedence over the global one in the function where it is declared.

Ex:

```
int count;
main()
{  count=10;
}
fun()
{ int count=0;
count=count+1;
}
```

External Declaration:

If a variable is declared after the main function, that variable cannot be accessed in main. This can be avoided by declaring the variable with the storage class extern.

Ex:

```
main()
{
 extern int y;    }
fun1()
{
  extern int y;
}
int y;
```

Note: The extern declaration does not allocate storage space fir variables. In case of arrays, the definition should include their size.

Storage: memory

Default initial value: zero

Scope: through out the program

Life: through out the entire program.

Static Variables:

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the keyword static like

```
static int x;
static float v;
```

A static variable may be either an internal type or an external type, depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extends up to the end of the function in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal state variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Example:
Write a program to illustrate the properties of a static variable.

```
main()
{
   int i;
   for(i=l; i<=3; i++)
   stat();
}
stat()
{
    static int x = 0;
    x = x+l;
    printf("x %d\n", x);
}
```

x=1 x=2 x = 3

Note: A static variable is initialized only once, when the program is compiled. It is never initialized again.
Storage: memory
Default initial value: zero
Scope: local to the block in which the variable is defined
Life: value of the variable persists between different function calls.

REGISTER VARIABLES:

It is possible to store a variable in one of the machine's registers, instead of keeping in the memory, since the register access is faster than a memory access. This can be possible by declaring the variable in the Register storage class.

```
Register int count;
```

Note : Only limited number of variables can be placed in the registers. C automatically converts register variables into non register variables once the limit exceeds.
Storage: c.p.u registers
Default Initial value: garbage value
Scope: local to the block in which the variable is defined.
Life: till the control remains within the block in which the variable is defined.

STRUCTURES

Introduction

The structure has many fields and each field may be different data type like integer, float, character, array even a structure. Normally, a structure is a heterogeneous data type where as the array is a homogeneous data type.

The Definition of Structure:

The collection of heterogeneous (different) types of data items is known as structure. When this is done, the entire collection can be referred to by a structure name. In addition, the individual components that are called fields (or) members can be accessed and processed separately.
There are two important distinctions between arrays and structures.

1. The elements of an array must all have the same type. In a structure the components or fields may have different types.
2. A component of an array is referred to by its position in the array where as each component of a structure has a unique name.

Structures and arrays are similar in that both must be defined with a finite number of components.

The Declaration of Structure:

The structure declaration contains two parts. One is define it and then declare the variables of that type.

Declaration:

The general format of a structure declaration is

```
Storage class struct tag_name
{
        data_type member1
        data_type member2
                .
                .
                .
        data_type membern;
};
```

template

- where as the storage class is an optional
- The key word structure and the braces are required.
- The user_defined_name (tagname) usually used, to declare structure variables but there are situations in which it is not required.
- The data type and members are any valid 'C' data objects
- The structure is terminated with a semicolon.
- Members of a structure themselves are not variables. They don't occupy any memory until they are associated with the structure variable.

Example:

```
struct date
{
```

```
        int day;
        char month[3];
        int year;
};
```

Declaring structure variables:

The structure variables can be declared by using the following syntax:

Struct st_name st_variable;

Example:

```
struct date today;
```

Note: The structure definition and declaration can be combined in one statement. Example is

```
struct date
{
        int day;
        char month[3];
        int year;
}dob, doj, today;
```

Type-Defined structures:

We can use the keyword typedef to define a structure as follows

```
typedef struct
  {
        data_type member1
        data_type member2

        ……..
        data_type membern
   } type_name;
```

the type_name represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

type_name variable1, variable2, . . .;

Accessing the members of a structure:

The structure members can be initialized by using the following syntax.

Structure_variable.member

Where **'.'** is known as member access operator.

Ex: today.day, today.month, today.year

Assigning values to structure members:

```
today.day=15
today.month=5
today.year=1991
```

Assigning values to structure elements or members through scanf( ) statement.

```
Scanf("%d", &today.day);
Scanf("%s", today.month);
Scanf("%d", &today.year);
```

Where the link between a member and a variable is established using the member operator'.'. Which is also known as 'dot operator' (or) period operator.

Example: Write a program to assign values to the members of a structure through scanf

```
main( )
{
struct student
{
        char name[10];
        int roll_no;
        char sex;
        float height;
}st1,st2;
        printf("Enter name,roll_no,sex,height);
        scanf("%s%d%c%f",name,&roll_no,&sex,&float);
        printf("%s%d%c%f",name,roll_no,sex,float);
}
```

INITIALIZING A STRUCTURE:

A structure can be initialized as the way of another data type in C. In keeping with the array analogy, a structure must be either static or external.

Example: 1

```
        static struct
        {
        int weight;
        float height;
        }
student={60,180.75};
```

Example: 2

```
        struct  st_record
      {  int weight;
         float height;
       };
static st_ record student={60,180.25}
```

Comparison of Structure variables: Two variables of the same structure type can be compared the same way as ordinary variables.
Example: person1 and person2 belongs to the same structure the following operations are valid.

```
person1=person2;
person1=person2;
person1 !=person2;
```

Example: Write a program to illustrate the comparison of structure variables.

```
Struct class
{
        int number;
        char name[20];
        float marks;
};
main( )
{
        int x;
        static struct class student1={101,"Rao",72.50};
        static struct class student2={102,"Raju",80.90};
        struct class student3;
        student3=student2;
        x=((student3.number==student2.number)&&(student3.marks==student2.marks)?1:0);
        if(x==1)
{       printf("\n student2 & student3 are same\n");
        printf("%d%s%f\n", student3.number,student3.name,student3.marks);   }
        else
        printf("\n student2 and student3 are different\n");   }
```

Arrays of Structures: It is well known that the array is a group of identical data which are stored in a consecutive memory locations in a common heading (or) common variable name. A similar type of structures placed in a common heading (or) a common variable name is called an **"Array of structures"**.
Initialization of arrays of structures:
Example:

```
Struct marks
{
        int sub1;
        int sub2;
        int sub3;
        };
        main( )
        {
        static struct marks student[3]={{45,65,50},{75,53,59},{57,69,70}}
```

student is an array of three elements std[0],student[1],student[2]; and initialize their numbers as follows.

```
student[0].sub1=45;
student[0].sub2=65;
student[0].sub3=50;   ----------      student[2].sub3=70;
```

Arrays with in structures: So far, the members of a structure have been declared as a ordinary data type such as char, int float only. Even a member of a structure can be array data type also.
Example:

```
Struct marks
{
        int  number;
        float subject[3];
}student[2];
```

Here structure variables are student[0], student[1]:
The member subject contains three elements, subject[0],subject[1],subject[2]: these elements can be accessed using appropriate subscripts.
Example:
student[0].subject[0]; student[0].subject[1]; student[0].subject[2]; ------ students[1].subject[2];
Structures within structures:
Structures within a structure means nesting of structures. (Nesting of structures is permitted in 'C'. It is permissible that a structure can also be used as a member of a structure.
Example:
Structure defined to store information about the salary of employee

```
struct employee
{
        char name[20];
        char department[10];
        int basic_pay;
struct
{
        int dearness;
        int house_rent;
        int city;
```

```
}allowance;
};
employee employee[100];
```

The members contained in the inner structure can be referred to as

```
employee.allowance.dearness;
employee.allowance.city;
```

it is an array employee[0].allowance dearness

- We can also use tagnames to define inner structures

Example:

```
I.      Struct pay
{
        int dearness;
        int house_rent;
};      int city;
II.     struct salary
{
        char name[20];
        char department[10];
        struct pay allowance;
};
struct pay arrears;
III   struct salary emp[100];
```

ARRAYS WITHIN A STRUCTURE

```
main( )
{
        struct marks
        {
        int sub[3];
        int total;
        };
        static struct marks student[3]={45,67,81,0,75,53,69,0,57,36,71,0};
        static struct marks total;
        int i,j;
        for(i=0;i<=2;i++)
        {
        for(j=0;j<=2;j++)
        {
        student[i].total+=student[i].sub[j]/*student marks total*/
        total.sub[j]+=student[i].sub[j] /*total marks in subjects*/
        }
        total.total+=student[i].total;
        }
        printf("STUDENT TOTAL \n\n");
        for(i=0;i<3;I++)
        printf("Student[%d]%d\n",i+1,student[i].total);
        printf("\n SUBJECT TOTAL \n\n");
        for(j=0;j<3;j++)
        printf("subject[%d] %d\n", j+1,total.sub[j]);
        printf("\n Grand total=%d \n", total.total);
}
```

Structures and Functions:

There are three methods by which the values of a structure can be transferred to from one function to another.

1. To pass each member of structure as an actual argument of the function call. This is most elementary and becomes unmanageable and inefficient when the structure size is large.
2. Passing of a copy of the entire structure to the called function. Here the actual argument cannot be changed if any change takes place in formal parameters or arguments.
3. The address location of the structure is passed to the called function. This most efficient method compared the second one.

The general format of sending a copy of a structure to the called function is

```
data_type function name(st_name)
        Struct_type st_name;
{
        -------
        -------
        -------
        return(exp);
}
```

Points to remember:

1. The called function must be declared for its type, appropriate to the data type it is expected to return.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
3. The return statement is necessary only when the function is returning some data.

4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the calling function.

UNIONS

It is well known that a structure is heterogeneous data type which allow to pack together different types of data values as a single unit. Union is also similar to a structure data type but the way the data are stored and retrieved is different.

Union stores values of different types in a single location. A union may contain one of many different types of values (as long as only one is stored at a time). The declaration and the usage of the union is the same as its structures. The union only holds a value for one data type if a new assignment is made. The previous values is forgotten.

The keyword union is used to declare the union data type. this is followed by a user_defined_name surrounded by braces which describes the member of the union.

The general format of the union is

```
Storageclass Union Userdefinedname
{
        datatype member1;
        datatype member2;

        datatype membern;
        };
```

Where the storage class is optional. The keyword Union and the braces are required. The datatype and members are any valid C data objects such as short int, float, char.

Example:1

```
Union simple
{
        int first;
        float second;
        char third;
        }one,two;
```

Example:2

```
Union value
      {
        int c;
        double d;
        };
        Union value x;
```

To access the Unions individual members as

Example:1

```
        One.first;
        One.second;
        One.third;
```

Example:2

```
        x.c
        x.d
```

Example: Write a program to initialize the member of a union and to display the contents of the union.

```
#include<stdio.h>
main( )
{
        union value
        {
        int i;
        float f;
        }
        union value x;
        x.i=10;
        x.f=-1456.45;
        printf("First member: %d \n", x.i);
        printf("Second member: %.2f \n", x.f);
        }
```

Output:

First member 3686:

Second member –1456.45

```
        x.bdate.day=12;
        x.bdate.month=7;
```

```
        x.bdate.year=1992;
        printf("First member %d \n", x.i);
        printf("Second member %0.2f\n", x.i);
        printf("Structure: \n");
        printf("%d/%d/%d \n", x.bdate.day,x.bdate.month,x.bdate.year);
        }
```

Output: First member 12
Second member 0.00

Only the recently assigned (accessed) float values are stored and displayed correctly and the integer values are displayed wrongly.

Note: The union only holds a value for one datatype of the large storage of their members.

A union may be a member of a structure, and the structure may be a member of a union moreover structures and unions may be freely mixed with arrays.

Example: Write a program to declare a member of a union as a structure datatype and to display the contents of the union.

```
#include<stdio.h>
main( )
{       struct date
        {  int day;
        int month;
        int year;
        union value
        { int i;
          float f;
          struct date bdate;
        };
        union value x;
        x.i=10;
        x.f=-1456.45
```

BIT FIELDS

A bit field is a special type of structure member, in that several bit fields can be packed into an int. while bit fields are variables, they are defined in terms of bit rather than character or integer. Bit fields are useful for maintaining single or multiple bit flags in an int without having to use logical AND and OR operations to set and clear them.

The formal declaration of the bit field is same as the declaration of a structure, but there is a difference in accessing and using a bit field in a structure. The number of bits required to a variable must be specified and followed by a colon while declaring a bit field. The bit fields must be signed or unsigned integers from 1 to 16 bits in length.

Advantages:

1. The bit field is very much useful with data items where only a few bits are required to indicate a true or false condition.
2. The bit field is used to save a memory space, as the number of bits required are declared for each variable in a structure.

The general format of the bit field declaration is as follows.

```
Struct tag_name
{
        datatype name1:bit_length;
        datatype name2:bit_length;
                    .
                    .
        datatype namen:bit_length;
        };
```

Example:

```
        struct date
        {
        unsigned int day:5;
        unsigned int month:4;
        unsigned int year:7;
        };
```

Where

- datatype is either int or unsigned int or signed int
- the bit_length is the number of bits used for the specified name
- signed field should have at least 2 bits
- the field name is followed by colon:
- the bit length is decided by the range of value to be stored
- the biggest value that can be stored is 2^n-1

Here is one restriction, one cannot take the address of a bit field which means that he cannot use scanf to read values into a bit field. Instead, one will have to read into a temporary variable and then assign its value to the bit field.

Note: It is possible to combine normal structure elements with bit field elements.

Write a program to initialize the member of a structure as a bit field data type and to display the contents of the structure.

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

```
main( )
{   struct value
    {
    int i;
    unsigned day:5;
    unsigned month:4;
    unsigned year:7
    float f;
    };
    struct value a={10,23,7,1992,-123.4};
    printf("Bit field initialization \n");
    printf("Integer value %d \n", a.i);
    printf("Date: %d/%d/%d\n", a.day,a.month,a.year);
    printf("Floating print value=%0.2f\n",a.f);
}
```

Output: bitfield initialization

integer value:10

date: 23/7/1992

floating point value=-123.4

Note: The bit fields may be accessed in a structure using a pointer operator.

POINTERS

Definition:

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

The pointer is a powerful technique to access the data by indirect reference because it holds the address of that variable where it has been stored in the memory.

The pointer has the following advantages.

- Provides functions which can modify their calling arguments.
- Supports dynamic allocation routines.
- Improves the efficiency of certain routines.

A pointer contains a memory address. If one variable contains the address of another variable, then the first variable is said to point to the second.

Sometimes, the pointer is the only technique to represent and to access the complex data structures in an easy way. In C, the pointers are distinct such as Integer Pointer, floating point number pointer, character pointer etc.

Pointer Declaration: A Pointer is a variable, which hold the memory address of an another variable. the pointer variable consists of two parts such as the pointer operator and the address operator.

Pointer operator: A pointer operator can be represented by the combination of *(asterisk) with a variable.

Example int *ptr;

Where ptr is a pointer variable which holds the address of an integer data type.

The general format of the pointer declaration is

data_type *pointer variable

Where data_type is a type of pointer variable such as integer, char and floating point number variable etc.

Example

```
float *fpointer;
double *dpoint;
char *mpoint1;
```

The base type of the pointer defines which type of variables the pointer is pointing to

Address Operator: An address operator can be represented by the combination of & (ampersand) with a pointer variable. The (&) is a unary operator that returns the memory address of its operand.

Example: m=&ptr;

The address operator '&' as an operator that returns the address of the variable following it.

Note: The operator '*' is the complement of '&'.

Pointer expressions: A pointer is a variable data type, so the general rule to assign its value to pointer is same as any other variable datatype.

Example:1
```
int x,y;
int *ptr1,*ptr2;
ptr1=&x;        /*The address of variable x is assigned to a pointer variable ptr1*/
```
Example:2 `y=*ptr1 /*The value pointing pointer variable ptr1 is assigned to the variable 'y'*/`

Example:3
```
ptr1=&x;
ptr2=ptr1;      /*The contents stored in ptr1 is assigned to ptr2*/
```

Invalid pointer declaration:

1.
```
int x;
int x_pointer;
x_pointer=&x;  /*pointer variable must have the prefix of  * */
```
2.
```
float  y;
float  *y_pointer;
y_pointer=y    /*while assigning value to the pointer variable the address operator (&) must be used along with the variable y */
```
3.
```
int x;
char *c_pointer;
```

```
    c_pointer=&x; /*mixed data type is not permitted */
4.  int x;
    int *x;
    x=&x; /*Redefined the variable*/
```

Example: Write a program to assign the pointer variable to another pointer and display the contents of both pointer variables and their address

```
#include<stdio.h>
main( )
{
        int x;
        int *ptr1,*ptr2;
        x=10;
        ptr1=&x;
        ptr2=ptr1;
        printf("value of x=%d\n", x);
        printf("value of ptr1=%d\n", *ptr1);
        printf("value of ptr2=%d\n",*ptr2);
        printf("The contents of ptr1=%u\n", ptr1);
        printf("The contents of  ptr2=%u\n",ptr2);
}
```

Output:

```
x=10
value of ptr1=10
value of ptr2=10
address of ptr1=6256
address of ptr2=6256
```

Pointer Arithmetic: There are four arithmetic operators that may be used with pointers such as addition(+),subtraction(-),incrementation(++),decrementation(--)

Printers are variables. They are not integers, but they can usually be displayed as unsigned integers. The conversion specifier for a pointer is added and subtracted.

Example: ptr++ causes the pointer to be incremented, but not by 1.

Ptr- - causes the pointer to be decremented, not by 1.

The general rule about pointer arithmetic is that pointer performs the operation in bytes of the appropriate storage class.

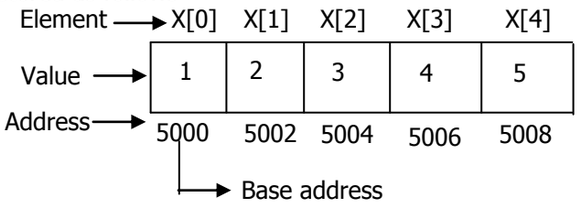
POINTERS AND ARRAYS:

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

ONE DIMENSIONAL ARRAY:

Example static int x[5]={1,2,3,4,5}

The five elements will be stored as follows



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

p=x;

This is equivalent to p= &x[0];

We can access every value of x using p++ to move from one element to another. The relation ship between p and x is given by

P=&x[0] (=5000)
P=&x[1] (=5002)
P=&x[2] (=5004)
P=&x[3] (=5006)
P=&x[4] (=5008)

The address of an element is calculated using its index and the scale factor of the data type.

Example: address of x[3]=base address +3 *(scale factor of int)
=5000+3x2=5006

Note: x[3] value can be accessed by *(p+3)

<u>Two-dimensional Array</u>:

Example:

	0	1	2
0	11	12	13
1	14	15	16

In a two dimensional array it is

&a[0][0]

Example:1

```
int value[][];
int *ptr;
ptr=value;
ptr+1=&value[0][1];
```

Example:2 float value[20][30];

float *ptr1;
ptr1=&value[0][0];
ptr1+4=&value[0][4];
ptr1+30=&value[1][0]; first element in second row

s[1][2] is expressed as *(*(s+1)+2) which is evaluated in the following order

s , s+1
*(s+1)+2
((s+1)+2)

Write a program to display the contents of a two dimensional array using a pointer arithematic.

/* Program to display contents of two dimensional array using pointer arithematic*/

Pointers and character strings:

Using pointers to the array and using pointer arithmetic usually performs many string operations in C, because strings tend to be accessed in a strictly sequential fashion. (Pointers use the obvious choice). Strings are one dimensional arrays of type **char**. In C, a string is terminated by null character ('\0'). String constants are written in double quotes.

Example: Write a program to sort names (strings) using pointers.

/* Program to sort names using pointers*/

```
#include<stdio.h>
#include<string.h>
#define SIZE 5
main()
{
  char *str[SIZE]={"Ramu","chandra","anil","Balu","Hussain"};
  int i;
  void sort();
  clrscr();
  sort(str);
  printf("sorted list is\n");
  for(i=0;i<SIZE;i++)
  printf("%s\n",str[i]);
  getch();
}
void sort( char *sstr[])      /*Function to sort Strings */
{
   char *temp;
   int i,j;
   for(i=0;i<SIZE-1;i++)
   for(j=i+1;j<SIZE;j++)
   if(strcmpi(sstr[i],sstr[j])>0)
   {
     temp=sstr[i];
     sstr[i]=sstr[j];
     sstr[j]=temp;
   }
}
```

Pointers and functions:

The pointers are very much used in a function declaration. Some times only with a parameter a complex function can easily be represented and accessed. The usage of the pointers in a function definition may be classified into two groups such as call-by-value and call- by-reference.

Difference:

In call by value the values are passed to function and the changes that are made in the function will not be reflected in main unless the values are returned.

In call by reference the addresses of the variables are passed instead of values and the changes that are made in the function are automatically reflected in main.

1. **Call by Value**:

The process of passing the values of actual arguments to the formal arguments is known as **call_by_value.**

Example:

/*Program to illustrate call by value display the contents of the variable actual arguments before or after function call */

```
#include<stdio.h>
main( )
{
        int i=20;
        printf("value of i before function call: %d \n", i);
        f(i);  /* function call */
        printf("\nvalue of i after function call: %d \n", i);
}
        f(x)      /*function destination*/
        int x;
        {
           x=5*x;
           printf("inside the function, value of i= %d\n",x);
           return(x);
```

```
}
```

Output:

value of i before function call=20
inside the function, value of i=100;
value of i after function call=20;

2. **Call by reference (Pointers as function arguments)**:

The process of calling a function using pointers (pointers as function arguments) to pass the address of variables is known as **call-by- reference**.

Example:

Write a program to illustrate the call by reference display the contents of the variable (actual any) before & after function call.

```
/* Program to illustrate the call by reference and display the actual arguments*/
#include<stdio.h>
main( )
{
        int i=20;
        printf("value of i before function call: %d\n", i);
        f(&i);
        printf("value of i after function call:%d\n", i);
}
        f(int *x);
        {
          int y;
          int y=5;
          *x=*x+5;
          printf("inside the function, value of x is: %d\n", *x);
        }
```

Output:

Value of i before function call=20
Inside the function, value of i=25
Value of i after function call=25

Pointers to functions:

A function, like a variable, has an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows.

```
type (*fptr) ( );
```

This tells the compiler that **fptr** is a pointer to a function which returns type value. The parenthesis around (*fptr) are necessary.

.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

Example:
```
double (*p1) ( ), mul( );
p1=mul;        Now p1 is pointing to the function mul( ).
```

To call the function mul, we may now use the pointer p1 with the list of parameters. That is,

```
(*p1) (x,y); which is equivalent to mul(x,y)
```

Note: A statement ike type *gptr( ); would declare gptr as a function returning a pointer to type.

Write a program to invoking function using pointer to a function.
Write a program passing function as a parameter to function

```
/*Invoking function using pointer to a function*/
main( )
{
        int display( );
        int (*func_ptr)( );
        func_ptr=display;
        ("\n address of function display is %u", function_ptr);
        (*func_ptr)( );
        }
        display( )/*function definition*/
        printf("\n long live viruses!!\n");
        }
```

Output:

Address or function display is 1125
long live viruses!!

Pointers and Structures:

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variable.

Example:

```
struct inventory
{
        char name[30];
        int number;
```

```
        float price;
} product[2], *ptr;
```

This statement declares product as an array of two elements, each of the type struct inventory and ptr as a pointer to data objects of the type struct inventory.

The assignment, ptr=product; /*it would assign the address of the Zeroth element of product to ptr, i.e. the pointer ptr will now point to product[0] . Its members can be accessed using the following notation.

```
ptr  -> name;
ptr -> number;
ptr -> price;
```

The symbol ` -> ` is called the arrow operator and is made up of minus sign and a greater than sign.

Note: ptr → is simply another way of writing product[0]

Example:
```
for(ptr=product; ptr<product+2;ptr++)
    printf("%s%d%f\n", ptr→name,ptr→number,ptr→price);
```

Note: we would also use the notation (*ptr). Number to access the member number.

Write a program to illustrate the use of structure pointers.

/*pointers to structure variables*/

```
struct invent
{       char *name[20];
        int number;
        float price;
};
main( )
{
        struct invent prodcut[3], *ptr;
        printf("INPUT \n");
        for(ptr=product:ptr<product+3;ptr++)
                scanf("%s%d%f", ptr→name,sptr→number,sptr→price);
        printf("OUTPUT\n");
        ptr=product;
        while(ptr<product+3)
        {
        printf("%20s %5d%10.2f\n", (*ptr).name,(*ptr),number,(*ptr).price);
        ptr++;
        }
}
```

Input:

```
washing_machine  5    7500
electronic_iron     12     350
two_in_one          7    1250
```

Output:

```
washing_machine  5    7500
electronic_iron     12     350
two_in_one          7    1250
```

<u>Pointers to pointers</u>:

An array of pointers is the same as pointer to pointers. The correct arrays of pointers is easy to indent and because the indices keep to meaning class. The Pointers to a pointer is a form of multiple of indirection's or a class of pointers.

In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the variable that contain the values desired. Multiple indirection's can be carried on to whatever extent desired, but there are few cases where more pointers to a pointer is needed or written. Excess indirection is difficult to follow and process to conceptual errors.

```
pointer---→variable
pointer---→pointer-----→variable
```

A variable i.e. a pointer to a pointer must be declared as such. This is done by placing an additional *infront of the variable's name.

Example: int **ptr2;

Where ptr2 is pointer which holds the address of another pointer.

Write a program to declare the pointer to pointer variable and to display the contents of these pointers.

/* program to illustrate pointers to pointers*/

```
#include<stdio.h>
main( )
{
        int value;
        int *ptr1;
        int **ptr2;
        value=120;
        printf(" value=%d\n", value);
        ptr1=&value;
        ptr2=&ptr1;
        printf("pointer1=%d\n", *ptr1);
        printf("pointer 2=%d\n", **ptr2);
```

output:

```
value=120
pointer1=120
pointer2=120
```

}

FILE MANAGEMENT IN C

File is a collection of data or set of characters may be a text or program. Basically there are two types of files used in the C language: **Sequential file** and **Random access file**. The sequential files are very easy to create than random access files. The data or text will be stored or read back sequentially. In the Random access file, the data can be accessed and processed randomly.

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

- naming a file
- opening a file
- reading data from a file
- writing data to a file
- closing a file

There are two distinct ways to perform file operations in C. the first one is known as the low level I/O and uses unix system calls. The second method is referred to as the high-level I/O operations and uses functions in C's standard I/O library.

Defining and Opening a File: If you want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include

- file name
- data structure
- purpose

1. **File name:**
   It is a string of characters that make up a valid file name for the OS.
2. **Data structure:**
   Data structure of a file is defined as FILE in the library of standard I/O function definitions.
3. **Purpose:**
   When we open a file, we must specify what we want to do with the file.

The general formats for declaring and opening a File

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

The first statement declares the variable **fp** as a **"Pointer to the data type FILE".** The second statement opens the file named filename and assigns an identifier to the FILE type pointer **fp.** [This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program].

The second statement also specifies the purpose of opening this file .The mode does this job. Mode can be one of the following.

MODE	DESCRIPTION
"w"	Means write to the FILE, if the file name exist already on the disk, then it is deleted and a new file will be created
"r"	Means read from a file. If the file doesn't exist, error is returned i.e., NULL is returned as the value of fopen.
"a"	Means append a file. New data are added to the end of the file
"r+"	Opens an existing file for updating
"w+"	Create a new file for reading and writing
"a+"	Opens for append, create it doesn't already exist

Example:

```
FILE fp1,fp2;
fp1=fopen("student","w");
fp2=fopen("results","r");
```

fclose( ):

The fclose( ) is used to close the file for reusage and the file pointer will be unlinked from the system name. Note that the argument for fclose( ) is a file pointer not a file name.

```
fclose(fptr);
```

The fclose( ) returns zero(0) if the close operation is successful other wise it returns '-1'

Example:

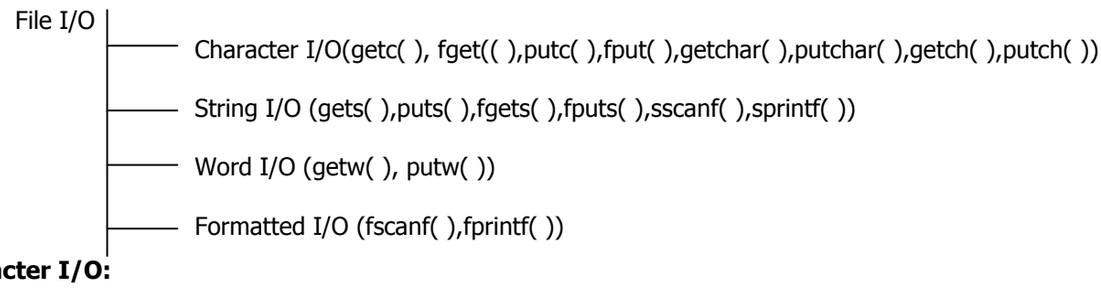
```
FILE *fp1,fp2;
fp1=fopen("student", "w");
fp2=fopen("results", "r");
-----
-----
fclose(fp1);
fclose(fp2);
```

Special names: The following is a list of the special names

1. Stdin — the name of a standard input file
2. Stdout — the name of a standard output file
3. Stderr — the name of a standard error file
4. EOF — the value returned by the read routines on the end of the file or error
5. NULL — the null pointer, returned by pointer valued functions, to indicate an error
6. BSIZE — the size in bytes (usually 1024) suitable for an I/O buffer supplied by the user

Input Output Operations of File:
SIMPLE (HIGH LEVEL) FILE OPERATIONS:

The simple file operation commands using different I/O facilities of <stdio.h> are



Character I/O:

1. **getchar( )**:

The getchar( ) is a function to use as an input statement to read any alphanumeric character from the standard input device (keyboard). Normally 'C' reads character as integers and to accept a character as a value for a variable of type integer.

2. **putchar( )**:

The putchar( ) is a complimentary function of the getchar( ). It is used to display any alpha numeric character on the standard out put device normally on the video screen. Normally, a character argument should be placed inside the putchar.

Example:
```
ch=getchar( );
putchar(ch);
```

Note: These functions actually don't exist in library

3. **getch( )**:

It is standard library function to read any alphanumeric character from the standard input devices. When getchar( ) is used ,it continues to access the keyboard until the carriage return[enter] key is pressed. While getch( ) stops accessing the keyboard as soon as any key is pressed.

4. **putch( )**:

The putch( ) is a function of <stdio.h> library to display any alphanumeric character on to the standard device normally, video screen.

5. **getc( )**:

The getc( ) is used to read a single character from a given file, and returns the value of the end of file[EOF] or an error is encountered.

The general format of getc( ) is:

```
ch=getc(fptr);
```

Where **fptr** is a file pointer of the file and **ch** is a variable receive the character.

6. **putc( )**:

The putc( ) is used to write a single character into a file. The general format of putc( ) is as follows.

```
Putc(ch,fptr);
```

Where **ch** is the character to be written and **fptr** is a file pointer to the file to receive a character.

Example:

Write a program to read the data from the key board write it to a file call INPUT, again read the data file INPUT and display contents on the screen.

/*Program to writing data & reading data from a file using getc( ) & putc( )*/

```
#include<stdio.h>
main( )
{
        FILE *fp;
        char c;
        printf("Data Input:\n");
        fp=fopen("input","w");
        While((c=getchar())!=EOF)
        {
                putc(c,fp);
        }
                fclose(fp);
        printf("\n Output Data:\n");
        fp=fopen("input","r");
        while((c=getc(fp))!=Eof)
        {
          putchar(c);
        }
        fclose(fp);
}
```

Output:

```
Data Input
This is a program to test the file handling
Features on this system ^Z
Output Data
This is a program to test the file handling
Features on this system
```

7. **fgetc( )**:

The fgetc( ) is also used to read a single character from a given file and returns the value end of the file EOF or an error is encountered. The getc( ) is a macro while fgetc( ) is true function in the <stdio.h> library.The operations otherwise are same.
The general format of the fgetc( ) function is as follows,

ch=fgetc(fptr);

Where **fptr** is a file pointer of the file, and ch is a variable to receive the character.

8. **fputc( )**:

The fputc( ) is also used to write a single character on a given file. The format of the fputc( ) is same as the putc(). The only difference is that the putc( ) is a macro and fputc( ) is a true function in <stdio.h> library. The general syntax of fputc( ) is

```
fputc(ch,fptr);
```

Where **'ch'** is a character to be written and **fptr** is a file pointer to the file to receive the character.

String I/O:

fgets( ):

The fgets( ) function is used to read a set of characters as a string from a given file and copies the string to a given memory location normally in an array. The general format of the fgets( ) function is

fgets(sptr,n,fptr);

Where **sptr** is a pointer to the location to receive the string, n is the count of the max number of characters to be in the string and **fptr** is the file pointer of the file to be read.

The **fgets()** function read n-1 characters or upto the first newline character, whichever occurs first. The function appends a null character('\0') to the last character read and then stores the string at the specified location. The function returns a null pointer value NULL if the end of the file or an error is encountered. Otherwise it returns the pointer **sptr.**

```
#include <stdio.h>
#define  MAX   200
main( )
{
        FILE *out_file;
        char a[max];
        ------
        ------
        if(fgets(a,MAX,out_file) !=NULLl)

        ------
        ------


}
```

fputs( ):

The fputs( ) function is used to write a string to a given file. The general format of the fputs( ) is as follows,

fputs(sptr,fptr);

Where **sptr** is a pointer to the string to be written and **fptr** is a file pointer to the file.
The fputs( ) function is normally used to copy strings from one file to another.

```
#include <stdio.h>
#define MAX 80
main( )
{
        FILE *in_file;
        char text[MAX];
        ------
        ------
        ------
        printf("Enter a sentence:\n");
        gets(text);
        fputs(text,in_file);
}
```

Write a program to read a string from the key board and store them in the given file & again read from the file and to display the contents of the file using fgets( ) & fputs( )

/*Program to read a string from keyboard & writing into file and again reading the data from the file*/

Example: WAP to Illustrate fgets( ) & fputs( )

```
/*Receives strings from keyboard and writes them to a file*/
#include<stdio.h>
main( )
{
        FILE *fp;
        char s[80];
        fp=fopen("poem.txt","w");
        if(fp==NULL)
        {
          puts("cannot open file \n");
          exit( );
```

```
        }
        printf("\n enter a few lines of text:\n");
        while(strlen(gets(s))>0)
        {
                fputs(s,fp);
                fputs("\n");
        }
                fclose(fp);
}
Enter a few lines of text:
        ----
        ----
        ----
```

/*Reads string from the file & displays them on screen*/

```
#include<stdio.h>
main( )
{
        FILE *fp;
        char s[80];
        fp=fopen("poem.txt","r");
        if(fp==NULL)
        {
          puts("cannot open file");
          exit( );
        }
        while(fgets(s,79,fp)!=NULL)
        printf("%s",s);
        fclose(fp);
}
```

Word I/O:

The <stdio.h> supports to read and write an integer value on a given file. The following functions are used to process the integer quantities. They are.

1. **getw( )**:

The getw( ) is used to read an integer value from a given file .It returns the next integer from the input file ,or EOF, an error encountered. The general format of the getw( ) is

```
getw(fptr);
```

where **fptr** is a pointer to a file to receive an integer value.

2. **putw( )**:

The putw( ) is used to write an integer quantity on to the specified file. The syntax of the putw() is

```
w=getw(fptr);
putw(w,fptr);
```

where **'w'** is an integer value to be return on a given file and **fptr** is a file pointer to a given file.

Example: **A file name DATA contains a series of integer numbers. Code a program to read these numbers and then write all add numbers to a file to be called ODD and all even numbers to a file to be called EVEN.**

/*HANDLING OF INTEGER DATA FILES */

```
#include<stdio.h>
main( )
{
        FILE *f1,*f2,*f3;
        int number i;
        printf("contents of DATA file:\n");
        f1=fopen("DATA", "w");
        for(i=0;i<30;i++)
        {
          scanf("%d", &number);
          if(number==-1)
                break;
          putw(number,f1);
        }
        fclose(f1);
        f1=fopen("DATA","r");
        f2=fopen("ODD","w");
        f3=fopen("EVEN","w");
        while((number=getw(f1))!=EOF)
        {
          if((number%2)==0)
            putw(number,f3)
          else
            putw(number,f2)
        }
        fclose(f1);
        fclose(f2);
        fclose(f3);

        f2=fopen("ODD","r");
        f3=fopen("EVEN","r")
        printf("\n contents of ODD file \n");
        while((number=getw(f2))!=EOF)
        printf("%4d \n", number);
        printf("\n contents of EVEN file \n");
        while((number=getw(f3))!=EOF)
        printf("%4d ", number);
        fclose(f2);
        fclose(f3);
} /*closing of main*/
```

Output:

```
Contents of data file
11 16 7 28  -17  -1
Contents of add file
11  7  17
Contents of even file
```

16 28

Formatted I/O :

The stdio.h library provides all types data handling features for the requirement of a programmer. Sometimes (one) it may be required to store and retrieve; only formatted data. The format that can also be processed with a file handling.

The following functions are used to access and process the formatted data.

fscanf( ),fprintf( );

fscanf( ): The fscanf( ) function is used to read a formatted data from a specified file. The control string to read a formatted data is same as the scanf( ) function. The scanf( ) function reads from the KB while fscanf( ) reads from a given file.

The general syntax of the fscanf( ) function is as follows.

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

where **fp** is a file pointer to receive a formatted data. The control string contains the format of data being received. list is a list of variables to be read from the given file.

Example:

```
#include<stdio.h>
main( )
{
        FILE *in_file;
        int x,y;
        in_file=fopen("name","r");
        while(fscanf(in_file,"%d%d", &x,&y) !=EOF)
        {
           ----------
           ----------
        }
           ----------
        fclose(in_file);
}
```

fprintf( ): The fprintf( ) function is used to write a formatted data into a given file. The printf( ) function is used to write or display the formatted data on the video screen while the fprintf( ) is used to write on a specified file. The control string is same for both printf & fprintf( ) functions.

The general format of the fprintf( ) is as follows.

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

Where

fp is a file pointer to write a formatted data, the control string contain the format of data being extract .list is the variable list to be written on to the file.

Example:

```
#include<stdio.h>
main( )
{
        FILE *infile;
        int x,y;
        infile=fopen(name,"w");
          ____________
          ____________
        fprintf(in_file,"%d%d\n",x,y);
          ____________
          ____________
        fclose(in_file);
}
```

Write a program (illustrate the fscanf( ) & fprintf( )) to create the employee data file with the following fields emp.name, age & basic_salary.

/*Creation of employee data file*/

```
#include<stdio.h>
main( )
{
        FILE *fp;
        char another='y';
        char name[40];
        int age;
        float bs;
        fp=fopen("Employee.dat","w");
        if(fp==NULL)
        {
          puts("cannot open file");
          exit( )   /*terminates the process*/
        }
        while(another='y')
        {
          printf("\n Enter name,age and basic salary\n");
          scanf("%s%d%f", name,&age,&bs);
```

```
            fprintf(fp,"%s%d%f\n", name,age,bs);
            printf("Another Employee(Y/N):");
            fflush(stdin); /*It is designed to 'flush out' any data remaining in buffer*/
            another=getch() /*read a character from kb and echoes it*/
          }
          fclose(fp)
}
```

Output:

```
Enter name, age and basic salary
  Anil 34,1550.55
another employee(Y/N): y
 Sanjay 35 1660.00
another employee(Y/N): n
```

To access the above file using fscanf( )

```
/*Accessing of employee data file*/
#include<stdio.h>
main( )
{
          FILE *fp;
          char name[40];
          int age;
          float bs;
          fp=fopen("Employee.dat","r");
          if(fp==NULL)
          {
          puts("cannot open file");
          exit( );
          }
          while((fscanf(fp,"%s%d%f", name,&age,&bs))!=EOF)
          {
          printf("\n %s%d%f", name,age,bs);
          }
          fclose(fp);
}
```

Output:

```
anil        34     1550.550000
sanjay    24     1200.000000
```

BLOCK READ/WRITE

fread( ); fwrite( )

If large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions fread( ) and fwrite( ) which store the numbers in binary format. (It means each number would occupy same number of bytes on disk as it occupies in memory).

fread( ):

The fread( ) function is used to read a block of binary data from a given file. It is also used to read one or more structures from a given file and copies them to given memory location.

The general format of the fread( ) function is as follows

```
fread(ptr,size,nitems,fp)
```

where

ptr - is a pointer to the location to receive the structure
size – is the size(in bytes) of each structure to be read
nitems – is the number of structures to be read
fp – is a file pointer to the file to be read

Example:

```
#include<stdio.h>
main( )
{
          FILE *outfile;
          struct database
          {
            char name[20];
            int age;
          };
          struct database person;
            ____________
            ____________
          fread(&person,size of(person),1,outfile);
            ____________
            ____________
          fclose(outfile);
}
```

Note: fread( ) function doesn't explicitly indicate errors so the feof( ) and ferror( ) functions should be used to detect end of the file or errors.

fwrite( ):

fwrite( ) function is used to write a binary data to a given file. It is also used to write one or more structures to a given file.

The general format of the fwrite( ) function is as follows

fwrite(ptr,size,nitems,fp)

where

ptr – is a pointer ,the first structure to be written
size – is the size (in bytes) of each structure
nitems – is the number of structures to be written
fp – is the file pointer to the file

The **ptr** may be pointer to a variable of any types. The size should give the number of bytes in each item to be written. The fwrite( ) function always returns the number of items actually written to the file whether a not of end of the file or an error is encountered.

Example:

```
#include<stdio.h>
main( )
{
        FILE *outfile;
        struct database
        {
          char name[20];
          int age;
        }
        struct database person;
             _____________
             _____________
        fwrite(&person,sizeof(person),1,outfile);
             _____________
             _____________
        fclose(outfile);
}
```

sizeof statement:

The sizeof is a keyword and not a function although it looks like a function in whole. The general syntax of the **sizeof** statement is

x= sizeof(item);

where x is a variable to which the size is to be assigned and item is an object (simple variable, constant or struct) one wants to know the sizeof.

Example:1

```
int x,y;
x=sizeof(y);
```

Example:2

```
struct sample
{
int x;
float y;
char ch;
char name[20];
}test;
int num;
num=sizeof(test);
```

Example:

Write a program to create employee data file with the following fields name, age, basic salary.

```
/* Receiving records from kb and writing them to a file in binary mode*/
#include<stdio.h>
main( )
{
        FILE *fp;
        char another='Y';
        struct emp
        {
          char name[40];
          int age;
          float bs;
        };
        struct emp e;
        fp=fopen("Emp.dat","wb");
        if(fp==NULL)
```

```
        {
          puts("cannot open file");
          exit( );
        }
        while(another='y')
        {
          printf("\n enter name,age,and basic salary");
          scanf("%s%d%f", e.name,&e.age,&e,bs);
          fwrite(&e,sizeof(e),l,fp);
          printf("Add another record(Y/N");
          fflush(stdin);
          another=getche( );
        }
        fclose(fp); }
```

output:

```
enter name,age,basic salary
suresh  24  1250.50
add another record (Y/N):Y
enter name age basicsalary
rajani  21  1300.60
```

Write a program to access the records from the above file

/*Read records from binary file and display them on visual display unit*/

```
#include<stdio.h>
main( )
{
        FILE *fp;
        struct emp1
        {
          char name[40];
          int age;
          float bs;
        };
        struct emp e;
        fp=fopen("emp.dat", "rb");
        if(fp==NULL)
        {
          puts("cannot open file",\n);
          exit( );
        }
        while(fread(&e,sizeof(e),1,fp)==1)
        {
          printf("\n %s%d%f",e.name,e.age,e.bs);
        }
        fclose(fp);
}
```

RANDOM ACCESS TO FILES:

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

ftell( ):

ftell() takes a file pointer and returns a number of type long, that corresponds to the current position. This function is useful in saving the current position. This function is useful in saving the current position of a file.

The general format of ftell() is as follows:

Where n would `n=ftell (fp);`
Give the relative offset (in bytes) of the current position. (This means n bytes have already been read(written))

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

The general syntax is

rewind(fp);

Example:

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

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

fseek: fseek function is used to move the file position to a desired location within the file. It takes the following form **fseek(fileptr, offset, position);** where

fileptr – is a pointer to the file concerned

offset – is a number or variable of type long. It specifies the number of positions (bytes) to be moved from the location specified by position. It may be + means forward, - means move backwards.

Position – it can take one of the following three values.

Value	**meaning**
0	beginning of file
1	current position
2	end of file

Example

fseek(fp,0L,0)	Go to the beginning
fseek(fp,0L,1)	Stay at the current position
fseek(fp,0L,2)	Go to the end of the file.
fseek(fp,m,0)	Move to (m+1)th byte in the file
fseek(fp,m,1)	Go forward by m bytes
fseek(fp,-m,1)	Go backward by m bytes from the current position
fseek(fp,-m,2)	Go backward by m bytes from the end

/*Example to ftell & fseek functions*/

```
#include<stdio.h>
main( )
{
        FILE *fp;
        long n;
        char c;
        fp=fopen("Random","w");
        while((c=getchar( ))!=EOF)
        {
            putc(c,fp);
        }
        printf("No of characters entered=%ld\n",ftell(fp));
        fclose(fp);
        fp=fopen("random","r");
        n=0L;
        while(feof(fp)==0))
        {
          fseek(fp,n,0);
          printf("position of % c is %ld\n", getc(fp),ftell(fp));
          n=n+5L;
```

```
        }
        putchar("\n");
        fseek(fp,-1L,2)
        do
        {
          putchar(getc(fp));
        }
        while(fseek(fp,-2L,1));
        fclose(fp);
}
```

output:

```
ABCD --------- Z
No. of characters entered=26
Position of A is 0
Position of F is 5
Position of Z is 25
Position of  is 30
ZYXW--------- A
```

feof( ):

This function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a non-zero integer value if all of the data from the specified file has be read, and returns zero otherwise.

Example

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

ferror( ):

This function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a non-zero integer if an error has be detected up to that point, during processing. It returns zero otherwise.

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

Command Line Arguments:

Every C program should have one main function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact main can take two arguments called **argc** and **argv** and the information contained in the command line is passed on to the program through those arguments, when main is called up by the system.

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

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

Example: c > program x_FILE y_FILE

Where **argc** is three

argv is an array of three pointers to strings as shown below

```
argv[0] --→ PROGRAM
argv[1] --→ X_FILE
argv[2] --→ Y_FILE
```

in order to access the command line arguments, we must declare the main function and its parameters as follows.

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

the first parameter in the command line is always the program name i.e. argv[0].

Example:

Write a program to copy one file into another file.

```
/* copy one file into another file */
#include<stdio.h>
main(int argc,char *argv[])
{
```

```
        FILE fs, *ft;
        Char ch;
        if(argc!=3)
        {
          puts("in sufficient arguments");
          exit( );
        }
        fs=fopen(argv[1],"r");
        if(fs==NULL)
        {
          puts("cannot open source file);
          exit( );
        }
        ft=fopen(argv[2],"w");
        if(ft==NULL)
        {
          puts("cannot open target file");
          fclose(fs);
          exit( );
        }
        while(1)
        {
          ch=fgetc(fs);
          if(ch==EOF)
              break;
          else
             fputc(ch,ft);
        fclose(ft);
}
```

output:

c> filecopy p1.c p2.c

filename fromfile tofile

Example :

Write a program to illustrate use of command line arguments

/* Program to illustrate use of Command Line arguments*/

```
#include<stdio.h>
main(int argc,char *argv[])
{
        int m=0;
        while(m<=argc)
        {
        printf("argv[%d]%s\n",m,argv[m]);
        m++;
        }
        printf("The value of argc=%d\n", argc);
        }
```

output:

temp1 "command line arguments" 6+7=13 9-3=6

argv[0] c:\tc\temp1.exe dos version 3 and above display the entire path

argv[1] comand line arguments

argv[2] 6+7=13

argv[3] 9-3=6

argv[4] null

the value of argc=4

Example:2

Write a program in c that reverses every word in the string input to it. The output should be indicate to the input

/*Program to illustrate the use of command line arguments*/

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

```
main(int argc,char *argv[])
{
        int i, length, m=1;
        while(m<argc)
        {
        length=strlen(argv[m]);
        for(i=length; i>=0;i--)
        printf("%c", argv[m][i]);
        m++;
        }
}
```

Output:

input string peep i modem peep
output string peep modem i peep

PREPROCESSOR:

The preprocessor is a program that modifies the C source program according to directives supplied in the program.

An original source program usually is stored in a file. The preprocessor doesn't modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. A preprocessor carries out the following actions on the source file before it is presented to the compiler.

These actions consist of

- replacement of defined identifiers by pieces of text
- conditional selection of parts of the source file
- inclusion of other files
- renumbering of source files and the renaming of the source files itself

The general rules for defining a preprocessor are as follows.

- all preprocessor directives begin with the sharp sign(#)
- they must start in the first column, there is no space between # and the directive
- the preprocessor directive is terminated not by a semicolon
- only one preprocessor directive can occur on a line
- the preprocessor directives may appear at any place in any source file; outside functions, inside functions or inside compound statements.

The C preprocessor is a simple macro processor that conceptually processes the source text of a C program before the compiler parses the source program. The common C preprocessor directives and their uses.

#include	Insert text from another file
#define	Define preprocessor macro
#undef	Remove macro definitions
#if	Conditionally include some text, based on the value of the constant expression
#ifdef	Conditionally include some text, with the sense of the test opposite that of #ifdef
#else	Alternative include some text, if the previous #if, #ifdef, or #ifndef test failed
#elif	Combination of #if and #else
#endif	Terminate conditional text
#line	Give a line number for compiler messages
#error	Terminate processing early

The preprocessor makes it easier to understand and portable from one machine to another machine.

MACROS:

The macro definition is the use of the #define directive to define constants in a c-program. The macros can be classified into two groups such as

1. simple macro definitions and
2. The macro with arguments

Advantages:

1. Easy to read write
2. Easy to check the string constants and debug
3. Easy to transfer from one machine to other
4. Gives good look to the program

Simple macro definition:

A macro is simply a substitution string that is placed in the program.

Example:

```
#define MAX 100
main( )
{
        char name[max];
        for(i=0;i<=max-1;++i)
        }
```

which is internally (used) replaced with the following program

```
main( )
{
        char name[100];
        for(i=0;i<=100-1;++i)
        }
```

and subsequently compiled.

The following #define statements are valid.

```
#define TRUE 1
#define FALSE 0
#define MAX_BLOCK 100
```

MACRO WITH PARAMETERS:

The more complex form of macro definition declares the names of formal parameters within parenthesis, separated by commas.

#define name(var1,var2……..varn) substitution string

Example :

```
#define PRODUCT(x,y)  (x*y)
#define  MAX(x,y) (x>y?(x):(y))
```