

MODULE 1: INTRODUCTION TO C LANGUAGE

Questions and Answers on Pseudocode solution to a problem

Q.1 What is algorithm? Write the characteristics of an algorithm. Give an example for algorithm.

Ans:

Definition of Algorithm

Algorithm is a step by step solution to a given problem. Sequence of steps written in order to carry out some particular task. Each step of an algorithm is written in english.

Various characteristics of an algorithm are

1. The algorithm must have definite start and end.
2. An algorithm may accept zero or more inputs
3. An algorithm must produce atleast one output
4. The steps of an algorithm must be simple, easy to understand and unambiguous.
5. Each step must be precise and accurate.
6. The algorithm must contain finite number of steps.

Example:

Algorithm 1: Algorithm for finding of area of triangle

Step 1: start

Step 2: Read base and height of triangle

Step 3: Calculate area of triangle

Step 4: print area of triangle

Step 5: stop

Algorithm 2: Algorithm for finding of sum and average of given three numbers

Step 1: start

Step 2: Read three numbers i.e. A, B and C

Step 3: find sum of three numbers i.e. $\text{sum} = A + B + C$

Step 4: find average of three numbers i.e. $\text{average} = \text{sum} / 3$

Step 4: print sum and average

Step 5: stop

Q. 2 What is pseudocode? Give an example for pseudocode.

Ans:

Definition of Pseudocode

Pseudocode is high level description of an algorithm that contains a sequence of steps written in combination of english and mathematical notations to solve a given problem.

Pseudocode is part english and part program logic.

Pseudocodes are better than algorithm since it contains ordered steps and mathematical notations they are more closer to the statements of programming language.

This is essentially an intermediate-step towards the development of the actual code(program). Although pseudo code is frequently used, there are no set of rules for its exact writing.

Example:

Pseudocode 1: Pseudocode for finding area of triangle

```
Pseudocode Area_of_Triangle
BEGIN
    READ base and Height
    CALCULATE Area_of_Triangle=(base*height)/2
    PRINT Area_of_Triangle
END
```

Pseudocode 2: Pseudocode for finding sum and average of three numbers

```
Pseudocode SUM_AVG
BEGIN
    READ A, B, and C
    CALCULATE sum=A+B+C
    CALCULATE average=SUM/3
    PRINT sum and average
END
```

Q.3 What is flowchart? List and define the purpose of symbols used to represent flowchart. Give an example for flowchart.

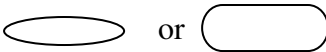
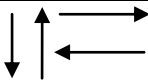


Ans:


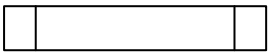
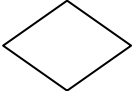
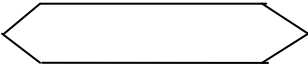
Definition of flowchart

Flowchart is a graphical or pictorial representation of an algorithm. Its a program design tool in which standard graphical symbols are used to represent the logical flow of data through a function.

Flowchart is a combination of symbols. They show start and end points, the order and sequence of actions, and how one part of a flowchart is connected to another.

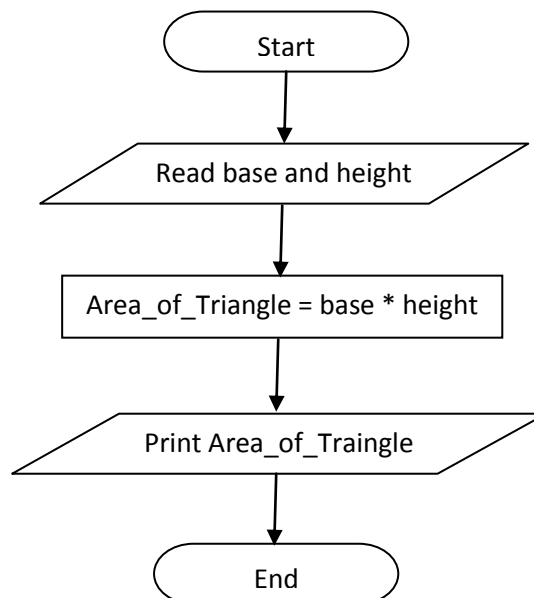
The symbols and their purpose in flowchart is given below.

Symbol	Name	Purpose
	Oval or Rounded Rectangle	Shows the Start or End of an algorithm
	Arrows	Shows the Flow of data
	Connector	Shows connecting points in an algorithm
	Parallelogram	Used to indicate Input or Output statement

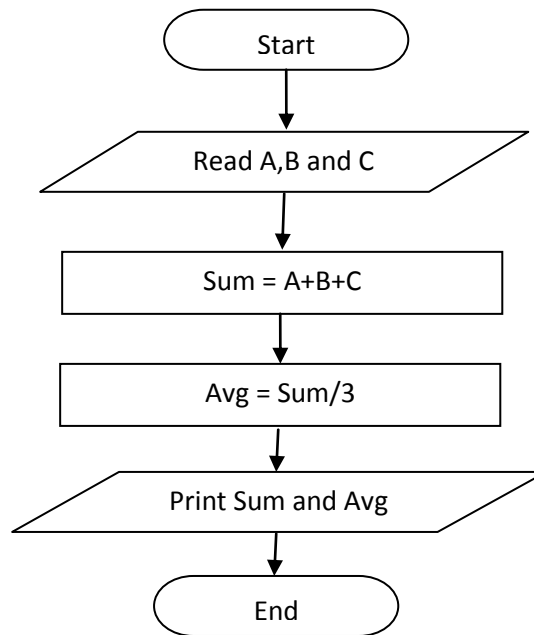
	Rectangle	Used to indicate assignment statement and executable statement
	Module Call	Used to represent function call
	Rhombus	Used to represent decision making statement
	Repitation	Used to represent looping statement

Example:

Flowchart 1: Finding the area of traingle



Flowchart 2: Finding sum and average of given three numbers



Questions and Answers on Basic Concepts in a C Program

Q.4 What are the *alphabets or character set* of C programming language?

Ans:

The characters that can be used to form words, numbers and expressions.

The characters in C are grouped into the following categories.

- **Letters:** Lowercase letters a,b,...,z and uppercase letters A,B,...,Z
- **Digits:** from 0 to 9
- **White spaces:** Space, tab(\t), newline(\n), carriage return(\r), etc
- **Symbols:**

~ tilde	! bang	% modulus
& ampersand	* star or asterisk	/ division
(left parentheses	{ left brace	[left bracket
) right parentheses	} right brace] right bracket
vertical bar	^ caret	' single quote
" double quotation	< less than	> greater than
: colon	; semicolon	\ back slash
. dot	, comma	? question mark
# hash or pound sign	_ underscore	= assignment

Q.5 What are *C-Tokens*? List and define different *C-Tokens* with examples.

Ans:

A C-Token is a smallest element of a C program. One or more characters are grouped in sequence to form meaningful words. These meaningful words are called C-Tokens.

The tokens are broadly classified as follows

- **Keywords:** Keywords are tokens which are used for their intended purpose only. Each keyword has fixed meaning or predefined meaning and that cannot be changed by user. Hence, they are also called as **reserved-words**.

ex: if, for, while, int, float, break, char ..etc.

- **Identifiers:** As the name indicates, identifier is used to identify various elements of program such as variables, constants, functions, arrays etc.

ex: sum, length, etc.

- **Constants:** A constant is an identifier whose value remains fixed throughout the execution of the program. The constants cannot be modified in the program.

ex: 10, 10.5, 'a', "sri", etc.

- **Operators:** An operator can be any symbol like + - * / that specifies what operation need to be performed on the data.

For ex: + indicates addition operation

* indicates multiplication operation

/ indicates division operation, etc.

- **Special symbols:** The symbols that are used to indicate array subscript, function call, block, etc.

ex: [], (), {}, etc.

Q. 6 What are *Keywords*? List all keywords and thier purpose/importance.

Ans:

Keywords are tokens which are used for their intended purpose only. Each keyword has fixed meaning or predefined meaning and that cannot be changed by user. Hence, they are also called **reserved-words**.

There are totally 32 keywords. All keywords are written in lowercase letters.

auto : The auto keyword declares automatic variables. Variables declared within function bodies are automatic by default.

break and continue: The *break* statement makes program jump out of the innermost enclosing loop (while, do, for or switch statements) explicitly. The *continue* statement skips the certain statements inside the loop.

switch, case and default : The switch and case statement is used when a block of statement has to be executed among many blocks/alternatives.

char : The char keyword declares a character variable.

const: An identifier can be declared constant by using const keyword.

do...while: do...while are used to repeat set of statements

double and float: Keywords double and float are used for declaring floating type variables.

if and else: if and else are used to make decisions.

enum: Enumeration types are declared in C programming using keyword enum.

extern: The extern keyword declares that a variable or a function has external linkage outside of the file it is declared.

for: for is used to repeat set of statements

goto: The goto keyword is used for unconditional jump to a labeled statement inside a function.

int: The int keyword declares integer type variable.

short, long, signed and unsigned: The short, long, signed and unsigned keywords are type modifiers that alter the meaning of a base data type to yield new type.

return: The return keyword terminates the function and returns the value.

sizeof: The sizeof keyword evaluates the size of a data (a variable or a constant).

register: The register keyword creates register variables which are much faster than normal variables.

static: The static keyword creates static variable. The value of the static variables persists until the end of the program.

struct: The struct keyword is used for declaring a structure. A structure can hold variables of different types under a single name.

typedef: The typedef keyword is used to define user defined name for a datatype.

union: A Union is used for grouping different types of variable under a single name for easier handling.

void: The void keyword indicates that a function doesn't return value.

volatile: The volatile keyword is used for creating volatile objects. A volatile object can be modified in unspecified way by the hardware.

Q. 7 What are identifiers? What are the rules for naming an identifier? Give examples.

Ans:

Identifier is used to name various elements of program such as variables, constants, functions, arrays, functions etc. An identifier is a word consisting of sequence of Letters, Digits or _(underscore).

Rules to Name identifiers are

1. The variables must always begin with a letter or underscore as the first character.
2. The following letters can be any number of letters, digits or underscore.
3. Maximum length of any identifier is 31 characters for external names or 63 characters for local names.
4. Identifiers are case sensitive. Ex. Rate and RaTe are two different identifiers.
5. The variable name should not be a keyword.

6. No special symbols are allowed except underscore.

Examples:

Valid identifiers:

Sum roll_no _name sum_of_digits
avg_of_3_nums

Invalid Identifiers and reason for invalid:

\$roll_no	-	Name doesn't begin with either letter or underscore
for	-	keyword is used as name
sum,1	-	special symbol comma is not allowed
3_factorial	-	name begins with digit as first letter

Q. 8 What are constants? List and explain different types of constants with examples.

Ans:

A constant is an identifier whose value remains fixed throughout the execution of the program. The constants cannot be modified in the program.

For example: 1, 3.14512, 'z', "hello"

Different types of constants are:

1) **Integer Constant:** An integer is a whole number without any fraction part.

There are 3 types of integer constants:

i) Decimal constants (0 1 2 3 4 5 6 7 8 9)	For ex: 0, -9, 22
ii) Octal constants (0 1 2 3 4 5 6 7)	For ex: 021, 077, 033
iii) Hexadecimal constants (0 1 2 3 4 5 6 7 8 9 A B C D E F)	
For ex: 0x7f, 0x2a, 0x521	

2) **Floating Point Constant:** The floating point constant is a real number.

The floating point constants can be represented using 2 forms:

i) **Fractional Form:** A floating point number represented using fractional form has an integer part followed by a dot and a fractional part.
For ex: 0.5, -0.99

ii) **Scientific Notation (Exponent Form):** The floating point number represented using scientific notation has three parts namely: mantissa, E and exponent.
For ex: 9.86E3 imply 9.86×10^3

3) **Character Constant:** A symbol enclosed within a pair of single quotes(') is called a character constant.

Each character is associated with a unique value called an ASCII (American Standard Code for Information Interchange) code.

For ex: '9', 'a', '\n'

ASCII code of A = 65, a = 97, 0 = 48, etc

4) **String Constant:** A sequence of characters enclosed within a pair of double quotes("") is called a string constant.

The string always ends with NULL (denoted by \0) character.

For ex: "9", "a", "sri", "\n", ""

5) **Escape Sequence Characters:** An escape sequence character begins with a backslash and is followed by one character.

- A backslash (\) along with some character give special meaning and purpose for that character.

- The complete set of escape sequences are:

- \b Backspace
- \f Form feed
- \n Newline
- \r Return
- \t Horizontal tab
- \v Vertical tab
- \\ Backslash
- \' Single quotation mark
- \" Double quotation mark
- \? Question mark
- \0 Null character

6) **Enumeration constants:** It is used to define named integer constants. It is set of named integer constants.

Syntax for defining enumeration constants

```
enum identifier { enumeration-list}; // enum is a keyword
```

Example:

1) enum boolean {NO, YES};

/* This example defines enumeration boolean with two constants, NO with value 0 and YES with value 1 */

In enumeration-list, the first constant value i.e. NO is ZERO since it is not defined explicitly, and then value of next constant i.e. YES is one more than the previous constant value (YES=1).

2) enum months {Jan=1, Feb, Mar, Apr, May, June, Jul, Aug, Sep, Oct, Nov, Dec};

/* This example defines enumeration months with 12 constants, Jan=1, Feb=2, Mar=3, Apr=4, ..., Dec=12 */

In enumeration-list, the first constant value i.e. Jan=1 since it is defined explicitly, and then value of next constants i.e. Feb, Mar, Apr, ..., Dec are one more than the previous constant value.

Q.9 What is datatype? List and explain different datatypes with examples.

Ans:

Datatype is an entity that defines set of values and set of operations that can be applied on those values.

Datatypes are broadly categorized as TWO types:

1) **Primitive Data Types:** Fundamental data types or Basic data types are primitive data types.

Examples: int, char, float, double, void

- 2) **Non-Primitive Data Types:** The derived or user-defined data types are called as non-primitive data types.

Examples: struct, union, pointers, etc.

- **int type:**

The int type stores integers in the form of "whole numbers". An integer is typically the size of one machine word, which on most modern home PCs is 32 bits. Examples of whole numbers (integers) such as 1,2,3, 10, 100... When int is 32 bits, it can store any whole number (integer) between -2147483648 and 2147483647. A 32 bit word (number) has the possibility of representing any one number out of 4294967296 possibilities (2 to the power of 32).

```
int numberOfStudents, i, j=5;
```

In this declaration it declares 3 variables, numberOfStudents, i and j, j here is assigned the literal 5.

- **char type**

The char type is capable of holding any member of the character set. It stores the same kind of data as an int (i.e. integers), but typically has a size of **one byte**. The size of a byte is specified by the macro CHAR_BIT which specifies the number of bits in a char (byte). In standard C it never can be less than 8 bits. A variable of type char is most often used to store character data, hence its name.

Examples of character literals are 'a', 'b', 'l', etc., as well as some special characters such as '\0' (the null character) and '\n' (newline, recall "Hello, World").

Note that the char value must be enclosed within single quotations.

```
char letter1 = 'a';    /* letter1 is being initialized with the letter 'a' */  
char letter2 = 97;     /* in ASCII, 97 = 'a' */
```

In the end, letter1 and letter2 both stores the same thing the letter 'a', but the first method is clearer, easier to debug, and much more straightforward.

- **float type**

float is short for floating point. It stores real numbers also, but is only one machine word in size. Therefore, it is used when less precision than a double provides is required. float literals must be suffixed with F or f, otherwise they will be interpreted as doubles. Examples are: 3.1415926f, 4.0f, 6.022e+23f. float variables can be declared using the float keyword.

- **double type**

The double and float types are very similar. The float type allows you to store single-precision floating point numbers, while the double keyword allows you to store double-precision floating point numbers – real numbers, in other words, both integer and non-integer values. Its size is typically two machine words, or 8 bytes on most machines. Examples of double literals are 3.1415926535897932, 4.0, 6.022e+23 (scientific notation).

- **void type**

It is an empty data type. It has no size and no value. It is used with functions which doesnot return any value, pointers,etc.

Range of values for char, int data type can be calculated with following formula

1) Range of Signed numbers (Both Positive and Negative numbers)

$$-2^{n-1} \quad \text{to} \quad +2^{n-1} - 1 \quad \text{where } n \text{ is number of bits}$$

Therefore,

- char - 1 byte (8 bits)
range of values = -2^{8-1} to $+2^{8-1} - 1 \Rightarrow -128$ to $+127$
- int - 2 bytes (16 bits in 16-bit OS)
range of values = -2^{16-1} to $+2^{16-1} - 1 \Rightarrow -32768$ to $+32767$

2) Range of Unsigned numbers (Only Positive Numbers)

$$0 \quad \text{to} \quad +2^n - 1 \quad \text{where } n \text{ is number of bits}$$

Therefore,

- char - 1 byte (8 bits)
range of values = 0 to $+2^8 - 1 \Rightarrow 0$ to 255
- int - 2 bytes (16 bits in 16-bit OS)
range of values = 0 to $+2^{16} - 1 \Rightarrow 0$ to $+65535$

Size of each data type and ranges of values is given below.

char	1 byte (8 bits)	range -128 to 127
int	16-bit OS : 2 bytes	range -32768 to 32767
	32-bit OS : 4 bytes	range -2,147,483,648 to 2,147,483,647
float	4 bytes	range 3.4E-38 to 3.4E+38 with 6 digits of precision
double	8 bytes	range 1.7E-308 to 1.7E+308 with 15 digits of precision

void generic pointer, used to indicate no function parameters, no return value etc.

Type Modifiers: Except for type void the meaning of the above basic types may be altered when combined with the following keywords.

- signed
- unsigned
- long
- short

Table: Use of modifiers to create modified data type

Primitive Data Type	Type Modifier	Modified Data Type
Char	Unsigned signed	unsigned char signed char
Int	Unsigned signed short long	unsigned int signed int short int short unsigned short int unsigned short

		signed short int signed short long int long unsigned long int unsigned long signed long int signed long
Float	N/A	N/A
Double	Long	long double

The **signed** and **unsigned** modifiers may be applied to types **char** and **int** and will simply change the range of possible values.

For example an unsigned char has a range of 0 to 255, all positive, as opposed to a signed char which has a range of -128 to 127.

An unsigned integer on a 16-bit system has a range of 0 to 65535 as opposed to a signed int which has a range of -32768 to 32767.

Note however that the default for type int or char is signed so that the type signed char is always equivalent to type char and the type signed int is always equivalent to int.

The **long modifier** may be applied to type int and double only. A long int will require 4 bytes of storage no matter what operating system is in use and has a range of -2,147,483,648 to 2,147,483,647.

A long double will require 10 bytes of storage and will be able to maintain up to 19 digits of precision.

The **short modifier** may be applied only to type int and will give a 2 byte integer independent of the operating system in use.

Examples:

short int, long int, long double, unsigned char, signed char, unsigned int, etc

Q.10 Give general structure of C program. Explain with an example program.

Ans:

A C program is essentially a group of instructions that are to be executed as a unit in a given order to perform a particular task.

Each C program must contain a `main()` function. This is the first function called when the program starts to run or execute.

A C program is traditionally arranged in the following order but not strictly as a rule.

/* Comment lines */

Preprocessor Directives

[Global Declaration]

int main()

{

[Local Declarations]

[Executable statements]

```
}
```

[User-defined Functions]

Example Program:

Consider first a simple C program which simply prints a line of text to the computer screen. This is traditionally the first C program you will see and is commonly called the “Hello World” program for obvious reasons.

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

```
int main()
```

```
{
```

```
    /* This is how comments are implemented in C    to comment out a block of text */
```

```
    // or like this for a single line comment
```

```
    printf( "Hello World\n" );
```

```
    return 0;
```

```
}
```

All C compiler include a library of standard C functions such as printf which allow the programmer to carry out routine tasks such as I/O operations, mathematical operations, string operations etc. but which are not part of the C language, the compiled C code merely being provided with the compiler in a standard form.

Header files must be included which contain prototypes for the standard library functions and declarations for the various variables or constants needed. These are normally denoted by a .h extension and are processed automatically by a program called the **Preprocessor** prior to the actual compilation of the C program.

Therefore, The line **#include <stdio.h>**

Instructs the pre-processor to include the file stdio.h into the program before compilation so that the definitions for the standard input/output functions including printf will be present for the compiler.

As you can see this program consists of just one function the mandatory **main** function. The parentheses, (), after the word main indicate a function while the curly braces, { }, are used to denote a block of code -- in this case the sequence of instructions that make up the function.

Comments are contained within a /* ... */ pair in the case of a block(multi-line) comment or a double forward slash, //, may be used to comment out single line.

The line **printf("Hello World\n ");**

is the only C statement in the program and must be terminated by a semi-colon. The statement calls a function called printf which causes its argument, the string of text within the quotation marks, to be printed to the screen. The characters \n are not printed as these characters are interpreted as special characters by the printf function in this case printing out a newline on the screen. These characters are called escape sequences in C and cause special actions to occur and are preceded always by the backslash character, \ .

Global and Local Declaration statements are used to declare global and local variables, arrays, functions, pointers, etc.

Questions and Answers on Declaration, Initialization and Print statements

Q. 11 What is Variable? What is the need for variables? What are the rules for naming variables. Give examples.

Ans:

Definition of Variable

Variable is an identifier used to name the memory location which holds the value. Variable is an entity whose value can be changed (Not Fixed) during the program execution.

Need for variables

It may help to think of variables as a placeholder for a value or data. You can think of a variable as being equivalent to its assigned value or data. In a C program, if a user wants to store and use any data then the user must create variables in a program to store the required data.

Rules for naming a Variable

1. The variables must always begin with a letter or underscore as the first character.
2. The following letters can be any number of letters, digits or underscore.
3. Maximum length of any identifier is 31 characters for external names or 63 characters for local names.
4. Identifiers are case sensitive. Ex. Rate and RaTe are two different identifiers.
5. The variable name should not be a keyword.
6. No special symbols are allowed except underscore.

Examples:

Valid identifiers:

Sum roll_no _name sum_of_digits
avg_of_3_nums

Invalid Identifiers and reason for invalid:

\$roll_no	-	Name doesn't begin with either letter or underscore
for	-	keyword is used as name
sum,1	-	special symbol comma is not allowed
3_factorial	-	name begins with digit as first letter

Q.12 What is Variable? How to Declare and initialize variables? Give examples.

Ans:

Definition of Variable

Variable is an identifier used to name the memory location which holds the value. Variable is an entity whose value can be changed (Not Fixed) or modified during the program execution.

Declaring a Variable

A variable thus has three attributes that are of interest to us: its **type**, its **value** and its **address** and before a C program can utilize memory to store a value or data it must claim the memory needed to store the values for a variable. This is done by **declaring variables**.

Declaring variables is the way in which a C program shows the number of variables it needs, name of the variables, range of values it can represent and how much memory they need.

Within the C programming language, when managing and working with variables, it is important to know the **type of variables** and **the size of these types**. Size of the datatypes can be hardware specific – that is, how the language is made to work on one type of machine can be different from how it is made to work on another.

All variables in C are typed. That is, every variable declared must be assigned with certain datatype.

General Syntax for declaring variables:

DataType Variable-list;

Where,

DataType can be => int, float, char, double, short, unsigned int, long double, etc

Variable-list can be => list of variables separated by comma.

DataType var1,var2,var3,...,varN;

Example:

```
int count;      /* It declares a variable count as integer type      */
float average; /* It declares a variable average as floating point type */
int number_students, i, j; /* In this declaration it declares 3 variables,
number_students, i and j. */
char code;      /* It declares a variable code as character type */
```

Variables can be declared as

1) Local variables:

When variables are declared inside functions as follows they are termed local variables and are visible (or accessible) within the function (or code block) only.

For Example,

```
int main()
{
    int i, j;
    ...
}
```

In the above example, the variables i and j are local variables, they are created i.e. allocated memory storage upon entry into the code block main() and are destroyed i.e. its memory is released on exit from the block. Therefore i and j are local to main().

2) Global Variables:

When variables declared outside functions they are termed as global variables and are visible throughout the program. These variables are created at program start-up and can be used for the entire lifetime of the program.

For Example,

```
int i;
int main()
{ ... }
```

In this example, `i` is declared as global variable and it is visible throughout the program.

Initialization or Assignment of a Variable

When variables are declared in a program it just means that an appropriate amount of memory is allocated to them for their exclusive use. This memory however is not initialised to zero or to any other value automatically and so will contain random values unless specifically initialised before use. Hence, variables may need to be assigned and store value in it. It is done using initialization.

General Syntax for initialization :-

Initialization of variable is done using assignement operator(=).

variable-name = expression ;

Where,

Expression can be - arithmetic expression, relational expression, logical expression, conditional expression, constant or an identifier that must reduce to a single value.

For Example :-

- 1) char ch;
 ch = 'a' ; /* in this example, a character constant 'a' is assigned to variable ch */
- 2) double d;
 d = 12.2323 ; /* in this example, a floating point constant is initialized to d */
- 3) int i, j, k;
 i = 20 ;
 j = 100;
 k = i + j; /* in this example, i is initialized to 20, j is initialized to 100 and k is initialized to addition of i and j i.e. 120 */

Combining declaration and initialization of variables

The variables can be declared and initialized at the time of creating i.e allocating memory storage for variables.

Syntax :- **DataType var-name = constant ;**

For Example :-

```
char ch = 'a' ;
double d = 12.2323 ;
int i, j = 20 ; /* note in this case i is not initialised */
```

Q.13 What are input and output functions? Explain printf() and scanf() functions with examples.

Ans:

Input Functions: The functions which help the user to feed some data into the program are called as input functions. When we are saying Input that means to feed some data into the program. This can be given in the form of file or from command line or from keyboard. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Examples: `scanf()`, `gets()`, `getchar()`, `fscanf()`, etc

Output Functions: The functions which help user to display or print output on screen or on paper using printer or in file are called as output functions. When we are saying Output that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen as well as you can save that data in text or binary files.

Examples: printf(), puts(), putchar(), fprintf(), etc.

printf(): print formatted

The printf() function is used for formatted output and uses a control string or format string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required.

General Syntax:

int printf(const char *format, ...);

Printf function writes output to the standard output stream stdout (computer screen) and produces output according to a format provided.

Printf function returns an integer constant which is the number of character displayed on screen successfully. **The return value can be ignored.**

Two ways to use printf()

- 1) N=printf("Text string");
- 2) N=printf("format string", variables-list);

Where,

Text string - text message to be displayed on screen. It is displayed as it is on screen.

Format string - it is a sequence of one or more format specifiers depending on type of data to be displayed.

Variable-list - list of one or more variables, expressions, or constants whose data will be displayed on screen

N - contains the return value of printf function. It is optional means that it can be omitted.

The more **common format specifiers** for displaying various types of data are given below

%c	character	%f	floating point
%d	signed decimal integer	%lf	double floating point
%i	signed integer	%e	exponential notation
%u	unsigned integer	%s	string
%ld	signed long	%x	unsigned hexadecimal
%lu	unsigned long	%o	unsigned octal
%%	prints a % sign		

For Example :-

```
int i ;
printf( "%d", i ) ;
```

In the above example, the format string is %d and the variable is i. The value of i is substituted for the format specifier %d which simply specifies how the value is to be displayed, in this case as a signed integer.


```
int N;  
N = printf("Hello\n");
```

In this example, the printf() prints the text string "Hello" on screen and returns the integer value 5 which will be stored in N.

Some more examples :-

```
int i = 10, j = 20 ;  
char ch = 'a' ;  
double f = 23421.2345 ;  
printf( "%d + %d", i, j ) ; /* values of i and j are substituted from the variable list in  
order as required */  
printf( "%c", ch ) ;  
printf( "%s", "Hello World\n" ) ;  
printf( "The value of f is : %lf", f ) ; /*Output as : The value of f is : 23421.2345 */  
printf( "f in exponential form : %e", f ) ; /* Output as : f in exponential form :  
2.34212345e+4
```

Scanf(): scan formatted

The scanf() function is used for formatted input and uses a control string or format string which is made up of a series of format specifiers to govern how to read out the values for the variables as required.

This function is similar to the printf function except that it is used for formatted input.

The space character or the newline character are normally used as delimiters between different inputs.

General Syntax of scanf():

```
int scanf("format string",Address-list);
```

where,

format string - it is a sequence of one or more format specifiers to read data as required.

Address-list - it is a list of address of one or more variables depending on number of format specifiers.

The format specifiers have the same meaning as for printf().

The more **common format specifiers** for reading various types of data are given below

%c	character	%f	floating point
%d	signed decimal integer	%lf	double floating point
%i	signed integer	%e	exponential notation
%u	unsigned integer	%s	string
%ld	signed long	%x	unsigned hexadecimal
%lu	unsigned long	%o	unsigned octal

For Example :-

```
int i, d ; char c ; float f ;  
scanf( "%d", &i ) ;  
scanf( "%d%c%f", &d, &c, &f ) ; /* e.g. type "10 x 1.234" */  
scanf( "%d:%c", &i, &c ) ; /* e.g. type "10:x" */
```

The & character is the address of operator in C, it returns the address of the memory location of the variable.

Note that while the space and newline characters are normally used as delimiters between input fields the actual delimiters specified in the format string of the scanf statement must be reproduced at the keyboard faithfully as in the case of the last example. If this is not done the program can produce somewhat erratic results!

The scanf function has a return value which represents the number of fields it was able to convert or read successfully. It can be ignored.

For Example :- num = scanf("%c %d", &ch, &i);

This scanf function requires two fields, a character and an integer, so the **value placed in num** after the scanf() call will be 2 if successful.

Questions and Answers on Operators and Expressions

Q.14 What are operators and expressions? Explain different types of operators and expressions with examples.

Ans:

One of the most important features of C is that it has a very rich set of built in operators including arithmetic, relational, logical, and bitwise operators.

- An **operator** can be any symbol like + - * / that specifies what operation need to be performed on the data.

For ex: + indicates addition operation

* indicates multiplication operation

- An **operand** can be a constant or a variable.
- An **expression** is combination of operands and operators that reduces to a single value.

For ex: Consider the following expression a + b here a and b are operands, while + is an operator.

Types of Operators and Expressions are:

- 1) Arithmetic Operators and Expressions
- 2) Assignment and Compound Assignment Operators
- 3) Increment and Decrement Operators
- 4) Relational Operators and Expressions
- 5) Logical Operators and Expressions
- 6) Conditional Operators and Expressions
- 7) Bitwise Operators and Expressions
- 8) Special Operators

1) Arithmetic Operators and Expressions:

Arithmetic operators are + - * / and %

Arithmetic expressions are expressions which contains only arithmetic operators in it.

+ performs addition - performs subtraction * performs multiplication

/ performs division operation

% modulus or reminder operation

For Example:-

```
int a = 5, b = 2, x ;
float c = 5.0, d = 2.0, f ;
x = a / b ; // integer division(5/2), therefore, x = 2.
f = c / d ; // floating point division(5.0/2.0), f = 2.5.
x = 5 % 2 ; // remainder operator, x = 1.
x = 7 + 3 * 6 / 2 - 1 ; // x=15,* and / evaluated ahead of + and -.
x = 7 + ( 3 * 6 / 2 ) - 1 ; // x = 15
x = ( 7 + 3 ) * 6 / ( 2 - 1 ) ; // changes order of evaluation, x = 60 now.
```

2) Assignment Operators: Used to initialize the variables with value.

Assignment operators is =

Compound assignment operators are +=, -=, /=, *=, %=. These are also called as shorthand operators.

Many C operators can be combined with the assignment operator as shorthand notation

For Example :- `x = x + 10 ;` can be replaced by `x += 10 ;`

Similarly for `-=`, `*=`, `/=`, `%=`, etc.

These shorthand operators improve the speed of execution as they require the expression, the variable x in the above example, to be evaluated once rather than twice.

3) Increment and Decrement Operators

There are two special unary operators in C, Increment `++`, and Decrement `--`, which cause the variable they act on to be incremented or decremented by 1 respectively.

For Example :- `x++ ;` /* equivalent to `x = x + 1 ;` */
`x-- ;` /* equivalent to `x = x - 1 ;` */

`++` and `--` can be used in Preincrement/Predecrement Notation(Prefix Notation) or Postincrement/Postdecrement Notation(Postfix notation).

In Preincrement/Predecrement Notation

- The value of the variable is either incremented or decremented first
- Then, will use updated value of the variable

But in Postincrement/Postdecrement Notation

- The value of the variable is used first
- Then the value of the variable is incremented or decremented

For Example :-

```
int i, j = 2 ;
i = ++j ; /* preincrement: therefore i has value 3, j has value 3 */
int i, j = 2 ;
i = j++ ; /* postincrement: therefore i has value 2, j has value 3 */
```

4) Relational Operators: The operators which are to compare or to check the relation between two or more quantities.

Relational Expressions are expressions that contains only relational operators.

The full set of relational operators are provided in shorthand notation

>	is greater than	>=	is greater than or equal to
<	is less than	<=	is less than or equal to

`==` is equal to `!=` is not equal to

The result of any relational expression is always either TRUE or FALSE.

Example 1:-

```
int x=2;
if ( x == 2 ) /* x ==2 is TRUE since x value is 2 */
    printf( "x is equal to 2\n" ); // prints x is equal to 2
else
    printf( "x is not equal to 2\n" );
```

Example 2:-

```
int x=5;
if ( x == 2 ) /* x ==2 is False since x value is 5 */
    printf( "x is equal to 2\n" );
else
    printf( "x is not equal to 2\n" ); // prints x is not equal to 2
```

5) Logical Operators: The operators which acts on only two logical values i.e. true and false.

Logical expressions are expressions that contains only logical operators. Usually to combine one or more relations these logical operators are used. Hence such expressions are called as relational logical expressions.

`&&` Logical AND `||` Logical OR
`!` Logical NOT

NOTE: Since C has no boolean datatype so to use these operators one must need to remember that

- ZERO(0) is always treated as FALSE and vice versa
- NON-ZERO (>0 or <0) value is always treated as TRUE. TRUE is always represented as 1.

For Example :-

```
if ( x >= 0 && x < 10 )
    printf( " x is greater than or equal to zero and less than ten.\n" );
```

For Example :- `2 > 1` -- TRUE so expression has value 1

`2 > 3` -- FALSE so expression has value 0

`i = 2 > 1` ; -- relation is TRUE -- has value 1, i is assigned value 1

NOTE: Every C expression has a value. Typically we regard expressions like `2 + 3` as the only expressions with actual numeric values. However the relation `2 > 1` is an expression which evaluates to TRUE so it has a value 1 in C. Likewise if we have an expression `x = 10` this has a value which in this case is 10 the value actually assigned.

6) Conditional Operators: the operators `?:` is called as conditional operator and it is also called as ternary operator since it operates on three operands.

Syntax: **expression1?expression2:expression3;**

In the above conditional expression, if expression1 is TRUE then returns the value of expression2 otherwise returns the value of expression3.

For example:-

```
5 > 12 ? 11: 12;                      // returns 12, since 5 not greater than 12.
```

```
10!=5 ? 4 : 3;      // returns 4, since 10 is not equal to 5.
12>8 ? a : b;       // returns the value of a, since 12 is greater than 8.
```

7) Bitwise Operators: These operators perform operations on each individual bit of the data value rather than on the usual data value. Hence the name bitwise.

These are special operators that act on char or int variables only.

& Bitwise AND	Bitwise OR
^ Bitwise XOR	~ Ones Complement
>> Shift Right	<< Shift left

Recall that type char is one byte in size. This means it is made up of 8 distinct bits or binary digits normally designated as illustrated below with Bit0 being the Least Significant Bit (LSB) and Bit 7 being the Most Significant Bit (MSB).

The value represented below is 13 in decimal.

Bit7(MSB)	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0(LSB)
0	0	0	0	1	1	0	1

An integer on a 16 bit OS is two bytes in size and so Bit15 will be the MSB while on a 32 bit system the integer is four bytes in size with Bit31 as the MSB.

Bitwise AND(&)

RULE : If any two bits in the same bit position are 1 then the resultant bit in that position is 1 otherwise it is zero.

For Example :- $1011\ 0010\ (178) \& \ 0011\ 1111\ (63) = 0011\ 0010\ (50)$

Bitwise OR (|)

RULE : If any two bits in the same bit position are 0 then the resultant bit in that position is 0 otherwise it is 1.

For Example :- $1011\ 0010\ (178) | \ 0000\ 1000\ (8) = 1011\ 1010\ (186)$

Bitwise XOR (^)

RULE : If the bits in corresponding positions are different then the resultant bit is 1 otherwise it is 0.

For Example :- $1011\ 0010\ (178) \wedge \ 0011\ 1100\ (60) = 1000\ 1110\ (142)$

Shift Operators, << and >>

RULE : These move all bits in the operand left or right side by a specified number of places.

Syntax :

```
variable << numberOfPlaces
variable >> numberOfPlaces
```

For Example :- $2 \ll 2 = 8$ i.e. $0000\ 0010$ becomes $0000\ 1000$

NB : shift left variable by numberOfPlaces multiplies variable by $2^{\text{numberOfPlaces}}$

i.e. resultant value = variable * $2^{\text{numberOfPlaces}}$

shift right variable by numberOfPlaces divides variable by $2^{\text{numberOfPlaces}}$

i.e. resultant value = variable / $2^{\text{numberOfPlaces}}$

Ones Complement (~)

RULE : bit 1 is set to 0 and bit 0 is set to 1. i.e. Each bit is negated.

For Example :- $1101\ 0011$ becomes $0010\ 1100$

Note: With all of the above bitwise operators we must work with decimal, octal, or hexadecimal values as binary is not supported directly in C.

The bitwise operators are most commonly used in system level programming where individual bits of an integer will represent certain real life entities which are either on or off,

one or zero. The programmer will need to be able to manipulate individual bits directly in these situations. A mask variable which allows us to ignore certain bit positions and concentrate the operation only on those of specific interest to us is almost always used in these situations. The value given to the mask variable depends on the operator being used and the result required.

For Example :- To clear bit 7 of a char variable.

```
char ch = 89 ; // any value
char mask = 127 ; // 0111 1111
ch = ch & mask ; // clears bit7 of a variable ch ;
```

For Example :- To set bit 1 of an integer variable.

```
int i = 234 ; // any value
int mask = 2 ; // a 1 in bit position 2
i = i | mask ; // sets the bit 1 of variable i
```

8) Special Operators: the following are called as special operators

- a. **sizeof** operator: used to determine the size, in bytes, of the variable or datatype. The sizeof operator returns the amount of memory, in bytes, associated with a variable or a type.

Syntax: sizeof (expression)

For Example:-

```
int x , size ;
size = sizeof(x) ;        // sizeof returns size of integer variable x
printf("The integer x requires %d bytes on this machine", size);
printf( "Doubles take up %d bytes on this machine", sizeof (double) ) ;
// prints the size of double data type in bytes
```

- b. **Type Cast** operator:

Type cast operator is used to convert one type of data to specific type of data in the expression which has the following syntax.

Syntax: (type) expression

For Example,

if we have an integer x, and we wish to use floating point division in the expression $x/2$ we might do the following

(float) x / 2

which causes x to be temporarily converted to a floating point value and then implicit casting causes the whole operation to be floating point division.

The same results could be achieved by stating the operation as

$x / 2.0$

which essentially does the same thing but the former is more obvious and descriptive of what is happening.

- c. **Comma operator** – used to separate the arguments in function header, variables in declaration, combine more than one statement in a single line, etc.

For Example,

```
int a,b,c;        // comma separates three variables
a=b,c=10,d=100;    // comma separates three statements
```

Q.15 What is type conversion? Explain different types of conversion.**Ans:**

Converting one type of data into another type of data is called as type conversion. What happens when we write an expression that involves two different types of data, such as multiplying an integer and a floating point number? To perform these evaluations, one of the types must be converted.

There are two types of Type conversions:

- 1) Implicit type conversion
- 2) Explicit type conversion

Implicit type conversion:

When the types of the two operands in a binary expression are different, C automatically converts one type to another. This is known as implicit type conversion.

Example:

```
char    c = 'A';  
int     i = 1234;  
long double d = 3458.0004;
```

```
        i = c;           // since i is integer type C automatically converts c='A' as c =  
65 and then assigns 65 to i
```

```
        d = i;           // since d is long double, value of i is converted to 1234.0 and it  
is assigned to d.
```

```
        X = d + i;       // X = double + int is converted to X = double + double, since d  
and i are of two different types(int is promoted to long double type), therefore result of X is  
4692.0004(3458.0004 + 1234.0)
```

Explicit type conversion:

Rather than let the compiler implicitly convert data, we can convert data from one type to another ourselves using **explicit type conversion**. Explicit type conversion uses the unary **cast** operator, which has a precedence of 14.

To cast data from one type to another, we specify the new type in parentheses before the value we want to be converted.

For example,

To convert an integer A to a float, we code the expression as given below

(float) A

One use of the cast operator is to ensure that the result of a divide is a real number. For example, if we calculated ratio of girls and boys (both are integer values) in a class without a cast operator, then the result would be an integer value. So to force the result in fractional form, we cast calculation as shown below.

Ratio = (float) No_of_Girls / No_of_Boys;

Or

Ratio = No_of_Girls / (float) No_of_Boys;

Q.16 What is precedence and associativity rule? Explain with precedence table.

Ans:

Precedence Rule: Precedence is used to determine the order in which operators with **different precedence** in a complex expression are evaluated.

Associativity Rule: Associativity is used to determine the order in which operators with the **same precedence** are evaluated in a complex expression.

Precedence is applied before associativity to determine the order in which expressions are evaluated. Associativity is then applied, if necessary.

When several operations are combined into one C expression the compiler has to rely on a strict set of precedence rules to decide which operation will take preference.

The precedence of C operators is given below.

Precedence	Operator	Associativity
16	() [] -> .(dot)	left to right
15	++ --(postincrement/decrement)	left to right
15	++ --(preincrement/decrement)	left to right
	! ~ +(unary) -(unary) * & sizeof	right to left
14	(type)	right to left
13	* / %	left to right
12	+ -	left to right
11	<< >>	left to right
10	< <= > >=	left to right
9	== !=	left to right
8	&	left to right
7	^	left to right
6		left to right
5	&&	left to right
4		left to right
3	? :	right to left
2	= += -= *= /= %= &= ^= = <<= >>=	right to left
1	,	left to right

Examples:

The following is a simple example of precedence:

$$2 + 3 * 4$$

This expression is actually two binary expressions, with one addition and one multiplication operator. Addition has a precedence of 12, multiplication has a precedence of 13 from above table. This results in the multiplication being done first, followed by the addition. The result of the complete expression is 14.

The following is a simple example of Associativity:

$$2 * 3 / 4$$

This expression is actually two binary expressions, with one multiplication and one division operator. But both multiplication and division has a precedence of 13(same precedence). Hence, apply the associativity rule as given in table above i.e left to right. This results in the multiplication being done first, followed by the division. The result of the complete expression is 1.

Question Appeared in Previous Year Question Papers

Q.17 WACP to which takes p,t,r as input and compute the simple interest and display the result.

Ans:

```
/* Program to find the simple interest */
#include<stdio.h>
int main()
{
    float p,t,r,si;
    printf("Enter p, t and r\n");
    scanf("%f%f%f",&p,&t,&r);
    si = (p*t*r)/100;
    printf("Simple interest = %f\n",si);
    return 0;
}
```

Q.18 What is the value of X in the following code segments? Justify your answers.

i) int a,b;	ii) int a,b;
float x;	float x;
a=4;	a=4;
b=5;	b=5;
x=b/a;	x = (float) b/a;

Ans:

i) x = b/a;
 x = 5/4; // x = int/int
 x = 1.0 // since 5/4(int/int) results as 1 but x is of type float so 1 is converted to float i.e. 1.0

ii) x = (float) b/a;
 x = (float) 5/4;
 x = 5.0/4; // 5 is converted to 5.0(float type) because of cast operator
 x = 5.0/4.0; // 4 is also converted to 4.0 because of implicit type conversion
 x = 1.25 // hence, the result is 1.25

Q.19 WACP to find diameter, area and perimeter of a circle.**Ans:**

```
/* program to find diameter, area and perimeter of a circle */
#include<stdio.h>
int main()
{
    float radius, diameter, area, perimeter;
    printf("Enter the radius\n");
    scanf("%f",&radius);
    diameter = 2 * radius;
    area = 3.142 * radius * radius;
    perimeter = 2 * 3.142 * radius;
    printf("Diameter of circle = %f\n",diameter);
    printf("Area of circle = %f\n",area);
    printf("Perimeter of circle = %f\n",perimeter);
    return 0;
}
```

Q.20 WACP to find area and perimeter of a rectangle.**Ans:**

```
/* program to find area and perimeter of a rectangle */
#include<stdio.h>
int main()
{
    float length, breadth, area, perimeter;
    printf("Enter length and breadth\n");
    scanf("%f%f",&length, &breadth);
    area = length * breadth;
    perimeter = 2 * (length + breadth);
    printf("Area of rectangle = %f\n",area);
    printf("Perimeter of rectangle = %f\n",perimeter);
    return 0;
}
```

Q.21 WACP to find sum and average of three given numbers.**Ans:**

```
/* program to find sum and average of three numbers */
#include<stdio.h>
int main()
{
    int a,b,c,sum;
    float avg;
    printf("Enter a, b and c\n");
```

```
scanf( "%d%d%d", &a, &b, &c);
sum = a + b + c;
avg = sum/3;
printf("Sum = %d\n",sum);
printf("Average = %f\n", avg);
return 0;
}
```

Q.22 WACP to convert temperature in Celsius into temperature in Fahrenheit.

Ans:

```
/* program to convert temperature in Celsius into temperature in Fahrenheit */
#include<stdio.h>
int main()
{
    float C,F;
    printf("Enter Temperature in Celsius\n");
    scanf( "%f", &C);
    F = 1.8 * C + 32;
    printf("Fahrenheit = %f\n", F);
    return 0;
}
```

Q.23 WACP to convert temperature in Fahrenheit into temperature in Celsius.

Ans:

```
/* program to convert temperature in Fahrenheit into temperature in Celsius */
#include<stdio.h>
int main()
{
    float C,F;
    printf("Enter Temperature in Fahrenheit \n");
    scanf( "%f", &F);
    C = (F - 32)/1.8;
    printf("Celsius = %f\n", C);
    return 0;
}
```

Q.24 Write a C program that computes the size of int, float, double and char variables.

Ans:

```
/* program to compute size of int, float, double and char variables */
#include<stdio.h>
int main()
{
    int i;
    float f;
    double d;
    char c;
    printf(" Size of integer variable = %d\n", sizeof(i));
    printf(" Size of float variable = %d\n", sizeof(f));
    printf(" Size of double variable = %d\n", sizeof(d));
    printf(" Size of character variable = %d\n", sizeof(c));
    return 0;
}
```

Q.25 Write an algorithm to find sum and average of N numbers.

Ans:

Algorithm for finding sum and average of N numbers

Step 1: start

Step 2: read the value of N

Step 3: initialize sum as 0

Step 3: until I less than or equal to N

Repeat

sum = sum + I

Done

Step 4: print sum

Step 5: stop

Q. 26 Write the C expression for the following:

$$\text{i) } A = \frac{5x+3y}{a+b}$$

$$\text{ii) } C = e^{|x+y-10|}$$

$$\text{iii) } D = \frac{e^{\sqrt{x}} + e^{\sqrt{y}}}{x \sin \sqrt{y}}$$

$$\text{iv) } B = \sqrt{s(s-a)(s-b)(s-c)}$$

$$\text{v) } E = x^{25} + y^{35}$$

$$\text{vi) } X = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Ans:

i) Given $A = \frac{5x+3y}{a+b}$

Corresponding C expression is $A = ((5*x) + (3*y)) / (a+b);$

ii) Given $C = e^{|x+y-10|}$

Corresponding C expression is $C = \exp(fabs(x+y-10));$

iii) Given $D = \frac{e^{\sqrt{x}} + e^{\sqrt{y}}}{x \sin \sqrt{y}}$

Corresponding C expression is

$$D = (\exp(\sqrt{x}) + \exp(\sqrt{y})) / (x * \sin(\sqrt{y}));$$

iv) Given $B = \sqrt{s(s-a)(s-b)(s-c)}$

Corresponding C expression is

$$B = \sqrt{s*(s-a)*(s-b)*(s-c)};$$

v) Given $E = x^{25} + y^{35}$

Corresponding C expression is

$$E = \text{pow}(x,25) + \text{pow}(y,35);$$

vi) Given $X = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$

Corresponding C expression is

$$X = (-b + \sqrt{b*b-4*a*c}) / (2*a);$$

Q. 27 Evaluate the following expression

Assume, `int i = 10, b=20,c=5,e=2;`

`float q =2.5, d=4.5;`

i) $a/b + (a/(2*b))$

ii) $f = ++a + b-- / q$ find value of f, a, and b

iii) $G = b \% a++;$ find value of G and a

iv) $H = b \% ++a;$ find value of H and a

v) $a+2 < b \parallel !c \ \&\& \ a==d \parallel a-2 \leq e$

Ans:

i) $a/b + (a/(2*b))$

`10/20 + (10/(2*20))` // substitute values of variables

`10/20 + (10/40)` // inner most parentheses is having highest precedence

`10/20 + 0` //parentheses is having highest precedence so evaluate it

//first, `10/40(int/int)` so result is 0

```
0 + 0          // division operators has higher precedence than +,
               //10/20(int/int) so result is 0
0      // overall result of expression
```

ii) **f = ++a + b-- / q find value of f, a, and b**

```
f = ++10 + 20-- / 2.5      // substitute values
f = ++10 + 20 / 2.5        // b = 19, since postdecrement(--) operator has
                           // higher precedence evaluate it first according to steps
f = 11 + 20 / 2.5          // a = 11, since preincrement(++) operator has
                           // higher precedence evaluate it first according to steps
f = 11 + 8.0               // 20/2.5(int/float) so result is floating point
                           // value(implicit type conversion)
f = 19.0                   // 11+8.0(int + float) so final result is also float
```

Therefore, f=19.0, a=11, b=19

iii) **G = b%a++; find value of G and a**

```
G = 20%10++;      // substitute values
G = 20%10         // a=11, postincrement(++) has higher precedence so
                  // evaluate it first according to steps
G = 0             // % operator results remainder hence answer is 0
```

Therefore, G = 0, a = 11

iv) **H = b%++a; find value of H and a**

```
H = 20%++10;      // substitute values
H = 20%11         // a=11, preincrement(++) has higher precedence so
                  // evaluate it first according to steps
H = 9             // % operator results remainder hence answer is 9
```

Therefore, H = 9, a = 11

v) **a+2 < b || !c && a==d || a-2 <=e**

```
10 + 2 < 20 || !5 && 10 == 4.5 || 10-2 <= 2      //substitute values
```

/* Since unary ! operator is having highest precedence so !5 => !true => false(0) */

```
10 + 2 < 20 || 0 && 10 == 4.5 || 10-2 <= 2
```

/* + and - operator have higher precedence and both have same precedence, so use associativity rule i.e. left to right, hence + is evaluated first */

```
12 < 20 || 0 && 10 == 4.5 || 10-2 <= 2
```

/* - has higher precedence so evaluate it first, 10-2 is 8 */

```
12 < 20 || 0 && 10 == 4.5 || 8 <= 2
```

/* < and <= have higher precedence and both have same precedence, so use associativity rule i.e. left to right, hence evaluate < first, 12 < 20 is true so result is 1 */

```
1 || 0 && 10 == 4.5 || 8 <= 2
/* <= has higher precedence so evaluate it first, 8 <= 2 is false so result is 0 */
1 || 0 && 10 == 4.5 || 0
/* == has higher precedence so evaluate it first, 10 == 4.5 is false, so result is 0 */
1 || 0 && 0 || 0
/* && has higher precedence so evaluate it first, 0 && 0 is 0 */
1 || 0 || 0

// use associativity rule of || operator i.e. left to right and evaluate 1 || 0 is 1 */
1 || 0
1 // result of complete expression
```

***** ALL THE BEST*****

MODULE 2: Branching and Looping

I. Statements in C are of following types:

1. **Simple statements:** Statements that ends with semicolon
2. **Compound statements:** are also called as block. Statements written in a pair of curly braces.
3. **Control Statements:** The statements that help us to control the flow of execution in a program.

There are two types of control statements in C

a. **branching** b. **looping**

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

- a. **Branching statements:** The statements that help us to jump from one statement to another statement with or without condition is called **branching statements**. These statements are also called as **selection statements** since they select some statements for execution and skip other statements. These statements are also called as **decision making statements** because they make decision based on some condition and select some statements for execution.

Branching is so called because the program chooses to follow one branch or another.

Branching Statements are of two types:

- i) **Conditional Branching statements:** The statements that help us to jump from one statement to another statement based on condition.

Example:

- a. simple if statement,
- b. if...else statement,
- c. nested if...else statement,
- d. else...if ladder (cascaded if statement), and
- e. switch statement

- ii) **Unconditional Branching statements:** The statements that help us to jump from one statement to another statement without any conditions.

Example:

- a. goto statement,
- b. break statement,
- c. continue statement and
- d. return statement

- b. **Looping statements:** The statements that help us to execute set of statements repeatedly are called as **looping statements**.

Example:

- a. *while* statement,

- b. *do...while* statement and
- c. *for* statement

i. Conditional Branching Statements

The statements that help us to jump from one statement to another statement based on condition.

a. Simple *if* Statement

This is basically a “one-way” decision statement.

This is used when we have only one alternative.

It takes an expression in parenthesis and a statement or block of statements. If the expression is **TRUE** then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be **TRUE** if its evaluated value is **non-zero**.

The **syntax** is shown below:

```
if (expression)
{
    statement1;
}
Next-Statement;
```

Where,

Expression can be arithmetic expression, relational expression, logical expression, mixed mode expression, constant or an identifier. Firstly, the expression is evaluated to true or false.

If the **expression** is evaluated to **TRUE**, then **statement1** is executed and **jumps** to Next-Statement for execution.

If the **expression** is evaluated to **FALSE**, then **statement1** is skipped and **jumps** to Next-Statement for execution.

The flow diagram (Flowchart) of simple if statement is shown below:

Example: Program to illustrate the use of if statement.

/* Program to find a given number is positive */

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

```
int main()
```

```
{
```

```
    int n;
```

```
    printf("Enter any non-zero integer: \n");
```

```
    scanf("%d", &n);
```

```
    if(n>0)
```

```
        printf("Number is positive number ");
```

```
    return 0;
```

```
}
```

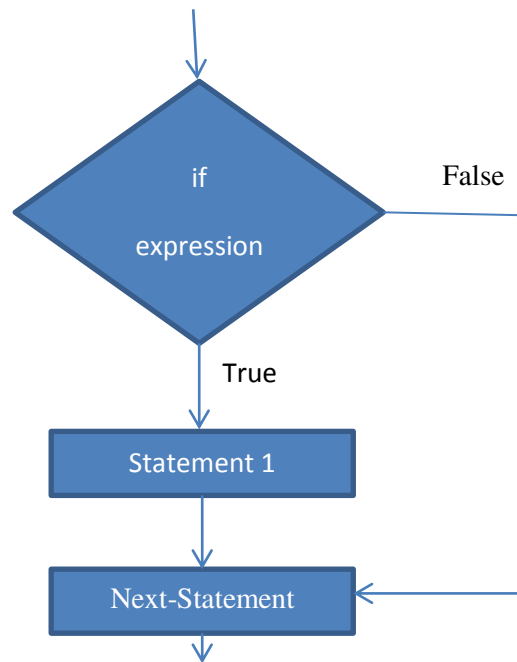
Output:

Enter any non-zero integer:

7

Number is positive number

In the above example, the program reads a integer number from user and checks is it greater than zero if true the it displays a message “Number is positive number” otherwise it does nothing. Hence, it considers only one alternative.

Flowchart of if statement**b. THE *if...else* STATEMENT**

This is basically a “two-way” decision statement. This is used when we must choose between two alternatives.

The syntax is shown below:

```
if(expression)
{
    statement1;
}
else
{
    statement2;
}
Next-Statement;
```

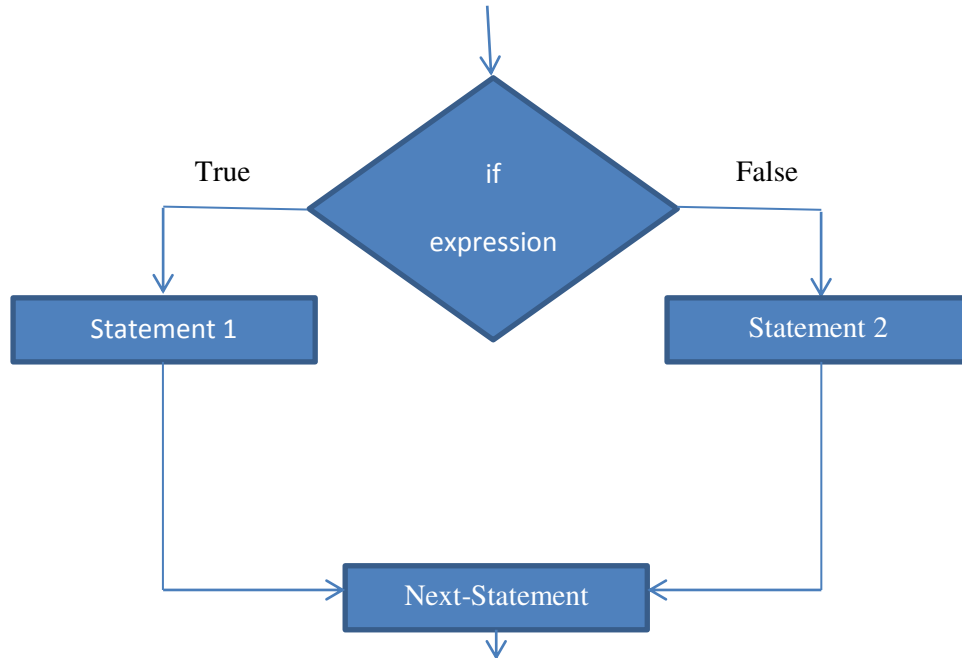
Where,

Expression can be arithmetic expression, relational expression, logical expression, mixed mode expression, constant or an identifier. Firstly, the expression is evaluated to true or false.

If the **expression** is evaluated to **TRUE**, then **statement1** is executed and **jumps** to Next-Statement (Skip statement2) for execution.

If the **expression** is evaluated to **FALSE**, then **statement2** is executed (**Skip statement1**) and **jumps** to Next-Statement for execution.

The flow diagram is shown below:



Example: Program to illustrate the use of if else statement.

```
#include<stdio.h>
int main()
{
    int n;
    printf("Enter any non-zero integer: \n") ;
    scanf("%d", &n)
    if(n>0)
        printf("Number is positive number");
    else
        printf("Number is negative number");
    return 0;
}
```

Output 1:

Enter any non-zero integer:

7

Number is positive number

Output 2:

Enter any non-zero integer:

-7

Number is negative number

In the above example, the program reads an integer number and prints "Number is positive number" if a given number "n" is greater than 0 otherwise it

prints "Number is negative number". Hence, it selects only one from two alternatives, so only it is called as two-way selection.

c. THE nested if STATEMENT

An if-else statement is written within another if-else statement is called nested if statement. This is used when an action has to be performed based on many decisions. Hence, it is called as **multi-way decision statement**.

The syntax is shown below:

```
if(expr1)
{
    if(expr2)
        statement1;
    else
        statement2;
}
else
{
    if(expr3)
        statement3;
    else
        statement4;
}
Next-Statement;
```

- Here, firstly **expr1** is evaluated to true or false.

If the **expr1** is evaluated to **TRUE**,
then

expr2 is evaluated to true or false.

If the **expr2** is evaluated to **TRUE**,
then

statement1 is executed.

If the **expr2** is evaluated to **FALSE**,
then

statement2 is executed.

If the **expr1** is evaluated to **FALSE**,
then

expr3 is evaluated to true or false.

If the **expr3** is evaluated to **TRUE**,
then

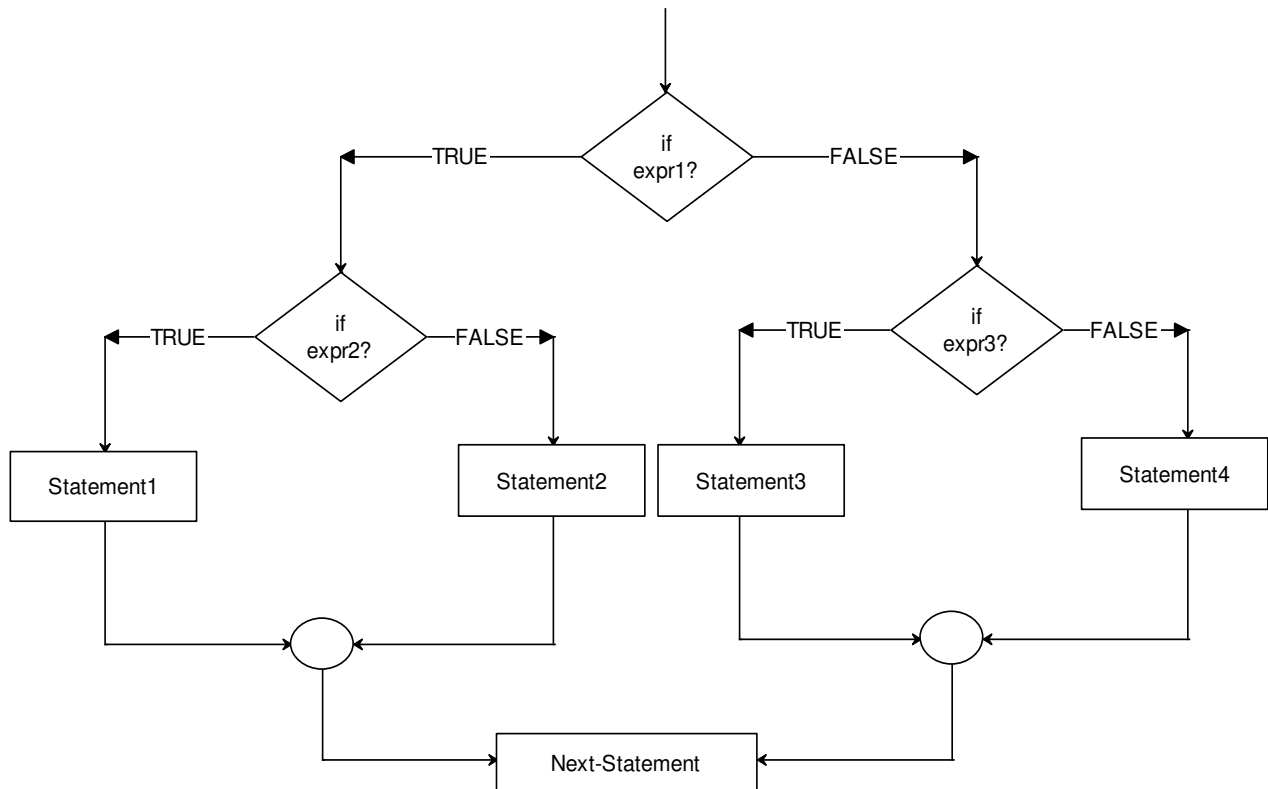
statement3 is executed.

If the **expr3** is evaluated to **FALSE**,
then

statement4 is executed.

The one of the drawback of this nested if statement is as the number of levels of nesting increases, the complexity increases and it makes difficult to understand and trace the flow of execution.

- The flow diagram is shown below:



Example: Program to select and print the largest of the 3 numbers using nested “if-else” statements.

```

#include<stdio.h>
int main()
{
    int a,b,c;
    printf("Enter Three Values: \n");
    scanf("%d %d %d ", &a, &b, &c);
    printf("Largest Value is: ") ;
    if(a>b)
    {
        if(a>c)
            printf(" %d ", a);
        else
            printf(" %d ", c);
    }
    else
    {

```

```
        if(b>c)
            printf(" %d", b);
        else
            printf(" %d", c);
    }
    return 0;
}
```

Output:

Enter Three Values:

17 18 6

Largest Value is: 18

d. THE CASCADED if-else STATEMENT (THE else if LADDER STATEMENT)

This is basically a “multi-way” decision statement. This is used when we must choose among many alternatives. This statement is alternative for nested if statement to overcome the complexity problem involved in nested if statement.

This statement is easier to understand and trace the flow of execution.

The **syntax** is shown below:

```
if(expression1)
{
    statement1;
}
else if(expression2)
{
    statement2;
}
else if(expression3)
{
    statement3
}
.
.
.
else if(expressionN)
{
    statementN;
}
else
{
    default statement;
}
Next-Statement;
```

The expressions are evaluated in order (i.e. top to bottom).

If an **expression** is evaluated to **TRUE**, then

- Statement associated with the expression is executed &
- Control comes out of the entire else if ladder

For example,

if **expression1** is evaluated to **TRUE**,

then **statement1** is executed and jumps out of entire else...if ladder.

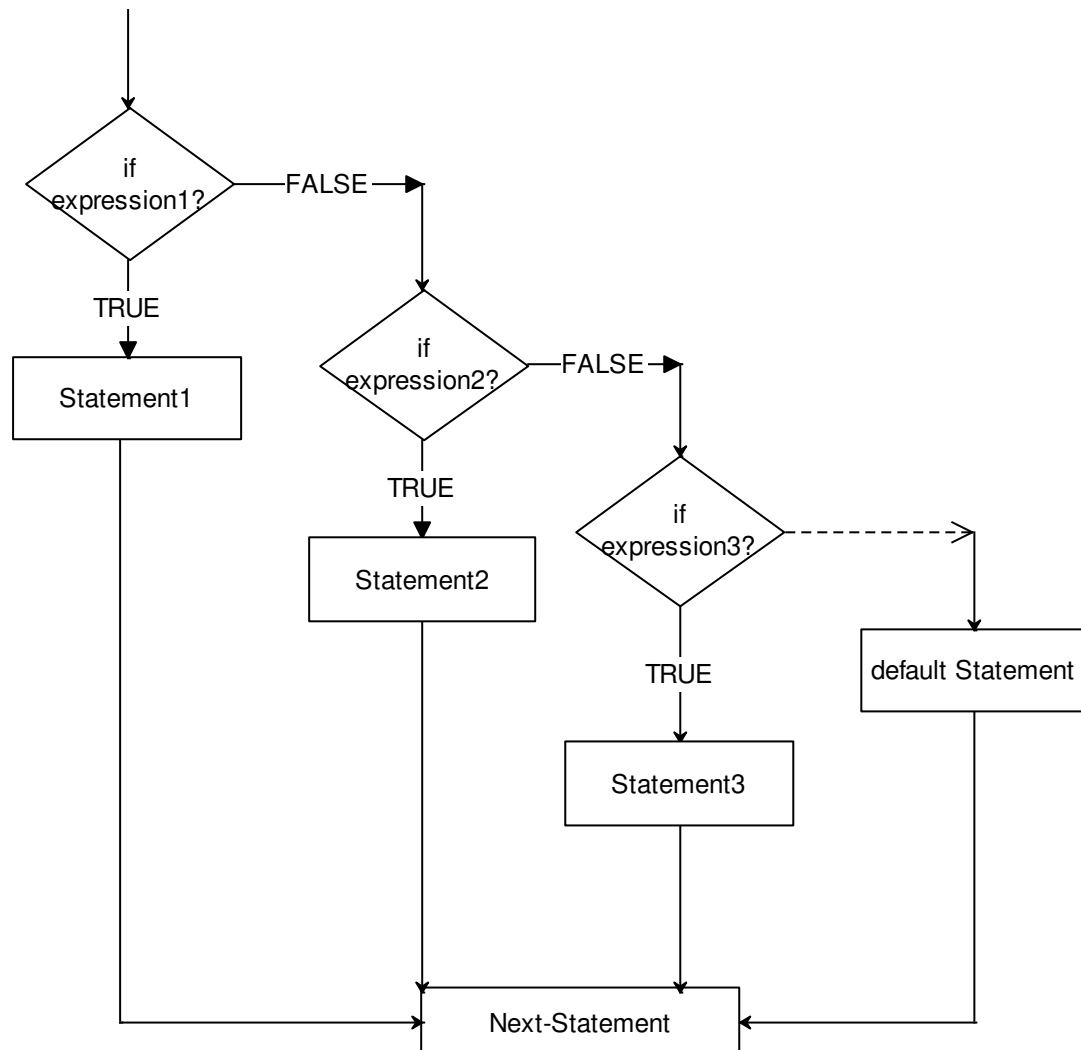
Illy, if **expression2** is evaluated to **TRUE**,

then **statement2** is executed and jumps out of entire else...if ladder.

If **all the expressions** are evaluated to **FALSE**,

then last statement (**default statement**) is executed.

Flowchart is shown below:



Example: Program to select and print the largest of the 3 numbers using “else...if ladder” statements.

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

```
int main()
```

```
{
```

```
    int a,b,c;
```

```
printf("Enter Three Values: \n");
scanf("%d %d %d ", &a, &b, &c);
printf("Largest Value is: ") ;
if ( a>b && a>c )
    printf(" %d ", a);
else if ( b>a && b>c )
    printf(" %d ", b);
else
    printf(" %d", c);
return 0;
}
```

Output:

Enter Three Values:

17 18 6

Largest Value is: 18

e. switch Statement:

This is a multi-branch statement similar to the else...if ladder (with limitations) but clearer and easier to code. This is used when we must choose among many alternatives. This statement is to be used only when we have to make decision using integral values (i.e., either integer values or characters).

Why we should use Switch statement?

1. One of the classic problems encountered in nested if-else / else-if ladder is called problem of Confusion.
2. It occurs when no matching else is available for if.
3. As the number of alternatives increases the Complexity of program increases drastically.

To overcome these, C Provides a multi-way decision statement called 'Switch Statement'

Syntax:

```
switch ( expression )
{
    case label1 :
        statement1 ;
        break ;
    case label2 :
        statement2 ;
        break ;
    ...
    default : statement ;
}
```

Where,

Expression can be either arithmetic expression that reduces to integer value, character constant, integer constant or an identifier of type integer.

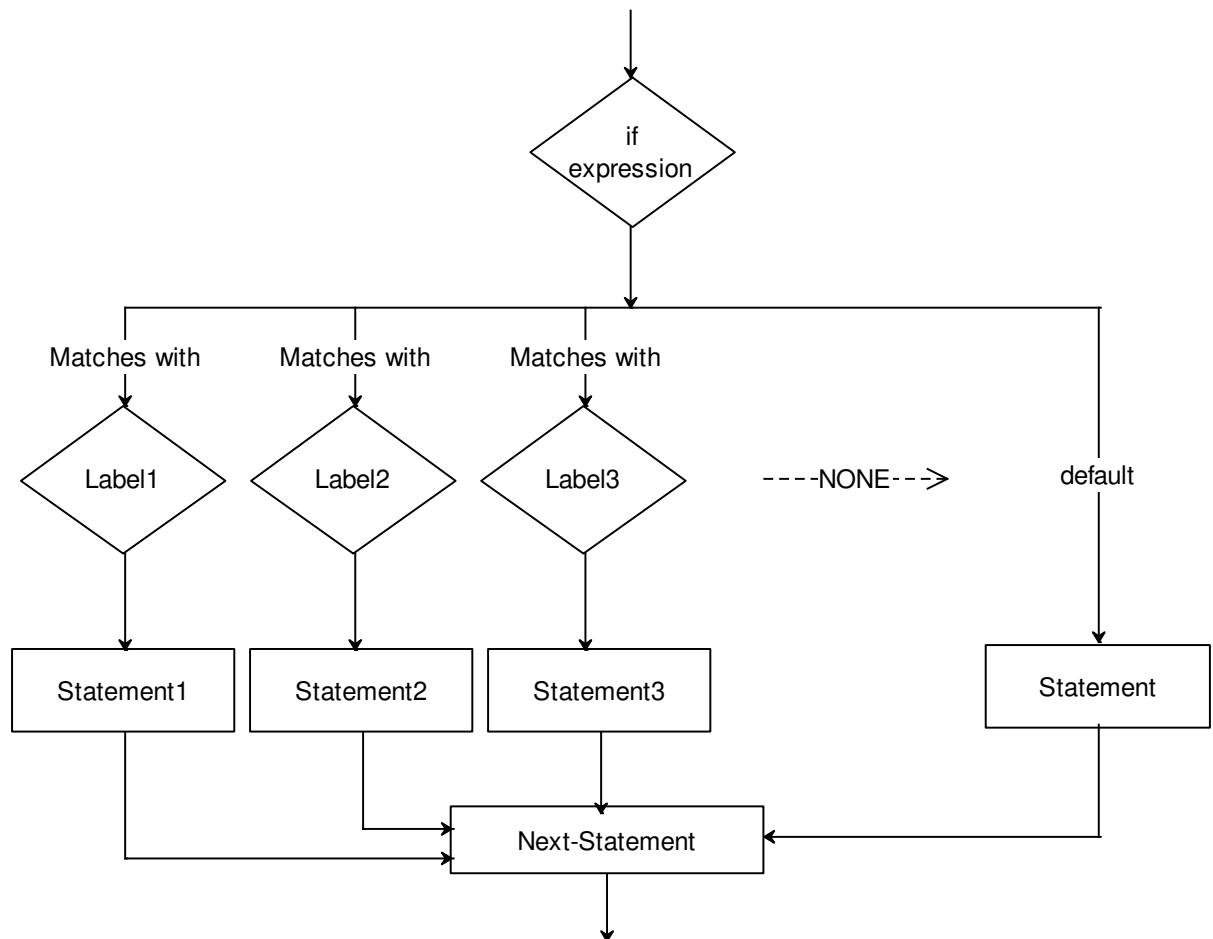
Labels can be either arithmetic expression that reduces to integer value, character constant or integer constant. But it cannot be an identifier.

The value of expression is tested for equality against the values of each of the labels specified in the case statements in the order written until a match is found. The statements associated with that case statement are then executed until a break statement or the end of the switch statement is encountered.

When a break statement is encountered execution jumps to the statement immediately following the switch statement.

The default section is optional -- if it is not included the default is that nothing happens and execution simply falls through the end of the switch statement.

The flow of switch statement is shown below:



The points to remember / Rules while using switch statement

- Can only test for equality with integer constants in case statements.
- All labels must be unique
- Character constants are automatically converted to integer.
- Floating point values are not allowed in expression and labels
- Break statement takes control out of the entire switch statement.
- Break Statement is Optional.

- Case Labels must end with Colon
- Case labels must have constants / constant expression
- Case label must be of integral Type (Integer, or Character)
- Switch case should have at most one default label
- default label is Optional
- default label can be placed anywhere in the switch
- Two or more cases may share one break statement
- Nesting (switch within switch) is allowed.
- Relational Operators and logical operators are not allowed in Switch Statement.
- Empty Switch case is allowed.
- Macro Identifiers are allowed as Switch Case Label.

Example:

```
#define MAX 2      /* defines a MAX with constant value 2, as pre-processor #define will replace occurrence of MAX by constant value i.e 2 therefor it is allowed. */
```

```
switch(num)
{
    case MAX:
        printf("Number = 2");
        break;
}
```

- Const Variable is allowed in switch Case Statement.
- ```
const int var = 2; /* it defines var as constant variable and whose value cannot be changed, hence this variable can be used as case label */
```

```
switch(num)
{
 case var: // var is constant hence it is allowed
 printf("Number = 2");
 break;
}
```

**For Example :- Program to simulate a basic calculator.**

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

```
int main()
```

```
{
 int num1, num2, result ;
 char op ;
 printf (" Enter number operator number\n") ;
 scanf ("%f %c %f", &num1, &op, &num2) ;
 switch (op)
 {
 case '+': result = num1 + num2 ;
```

```

 break ;
 case '-': result = num1 - num2 ;
 break ;
 case '*': result = num1 * num2 ;
 break ;
 case '%': result=num1%num2;
 break;
 case '/': if (num2 != 0.0)
 {
 result = num1 / num2 ;
 break ;
 }
 // else we allow to fall through for error message
 default : printf ("ERROR -- Invalid operation or division by 0.0") ;
}
printf("%f %c %f = %f\n", num1, op, num2, result) ;
return 0;
}

```

Note : The break statement need not be included at the end of the case statement body if it is logically correct for execution to fall through to the next case statement (as in the case of division by 0.0) or to the end of the switch statement (as in the case of default : ).

#### Output:

Enter number operator number

2 + 4

**2 + 4 = 6**

## II. Ternary operator (?:)

This operator operates on three operands hence the name ternary. This is a special shorthand operator in C and replaces if...else statement

Syntax of ternary operator is:

Expression1 ? expression2 : expression3;

If **expression1** is evaluated **TRUE**, then returns the result of expression2. i.e., it selects expression2 for execution and skips expression3.

If **expression1** is evaluated **FALSE**, then returns the result of expression3. i.e., it selects expression3 for execution and skips expression2.

Hence, it is same as TWO-WAY selection, it selects among two alternatives. Therefore any if...else statement can be replaced with this ternary operator.

For example,

```

if (expression1)
 expression2;
else
 expression3;

```

can be replaced with the more elegant form using ternary operator:

expression1 ? expression2 : expression3 ;

The ?: operator is a ternary operator in that it requires three arguments. One of the advantages of the ?: operator is that it reduces simple conditions to one simple line of code which can be thrown unobtrusively into a larger section of code.

**For Example :- to get the maximum of two integers, x and y, storing the larger in max.**

```
if (x >= y)
 max = x ;
else
 max = y ;
```

The alternative to this could be as follows

max = x >= y ? x : y ;

giving the same result and it is a little bit more short.

### Programming examples on branching statements and ternary operator:

1. Write a C program (WACP) to find largest of two numbers using if...else statement.

**Program:**

```
#include<stdio.h>
int main()
{
 int A, B, large;
 printf("Enter two numbers for A and B\n");
 scanf("%d%d", &A, &B);
 if(A>B)
 large=A;
 else
 large=B;
 printf("Largest of %d and %d = %d\n",A,B,large);
 return 0;
}
```

2. Write a C program (WACP) to find smallest of two numbers using if...else statement.

**Program:**

```
#include<stdio.h>
int main()
{
 int A, B, small;
 printf("Enter two numbers for A and B\n");
 scanf("%d%d", &A, &B);
 if(A<B)
```

```
 small=A;
 else
 small=B;
 printf("Smallest of %d and %d = %d\n", A, B, small);
 return 0;
}
```

### 3. WACP to find a given year is leap year or not using if...else statement.

#### Program:

```
#include<stdio.h>
int main()
{
 int year;
 printf("Enter a year\n");
 scanf("%d", &year);
 if(year%4 == 0 && year%100 != 0 || year%400 == 0)
 printf("%d is a Leap Year\n", year);
 else
 printf("%d is Not a Leap Year\n", year);
 return 0;
}
```

### 4. WACP to determine a given number is even or odd using if...else statement.

#### Program:

```
#include<stdio.h>
int main()
{
 int number;
 printf("Enter a number\n");
 scanf("%d", &number);
 if(number%2 == 0)
 printf("%d is a EVEN Number\n", number);
 else
 printf("%d is ODD Number\n", number);
 return 0;
}
```

**5. WACP to determine a given number is positive or negative using if...else statement.**

**Program:**

```
#include<stdio.h>
int main()
{
 int number;
 printf("Enter a number\n");
 scanf("%d", &number);
 if(number >= 0)
 printf("%d is a Positive Number\n", number);
 else
 printf("%d is Negative Number\n", number);
 return 0;
}
```

**6. Write a C program (WACP) to find largest of two numbers using ternary operator.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, large;
 printf("Enter two numbers for A and B\n");
 scanf("%d%d", &A, &B);
 large=(A>B) ? A : B ;
 printf("Largest of %d and %d = %d\n",A,B,large);
 return 0;
}
```

**7. Write a C program (WACP) to find smallest of two numbers using ternary operator.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, small;
 printf("Enter two numbers for A and B\n");
 scanf("%d%d", &A, &B);
 small=(A<B) ? A : B ;
 printf("Smallest of %d and %d = %d\n", A, B, small);
 return 0;
}
```

```
}
```

**8. WACP to find a given year is leap year or not using ternary operator.**

**Program:**

```
#include<stdio.h>
int main()
{
 int year;
 printf("Enter a year\n");
 scanf("%d", &year);
 (year%4 == 0 && year%100 != 0 || year%400 == 0) ? printf("%d is a Leap
Year\n", year) : printf("%d is Not a Leap Year\n", year);
 return 0;
}
```

**9. WACP to determine a given number is even or odd using ternary operator.**

**Program:**

```
#include<stdio.h>
int main()
{
 int number;
 printf("Enter a number\n");
 scanf("%d", &number);
 (number%2 == 0) ? printf("%d is a EVEN Number\n", number) : printf("%d
is ODD Number\n", number);
 return 0;
}
```

**10. WACP to determine a given number is positive or negative using ternary operator.**

**Program:**

```
#include<stdio.h>
int main()
{
 int number;
 printf("Enter a number\n");
 scanf("%d", &number);
 (number >= 0) ? printf("%d is a Positive Number\n", number) : printf("%d is
Negative Number\n", number);
 return 0;
}
```

**11. Write a C program (WACP) to find largest of three numbers using nested if statement.****PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, C, large;
 printf("Enter three numbers for A , B and C\n");
 scanf("%d %d %d", &A, &B, &C);
 if(A>B)
 {
 if(A>C)
 large=A;
 else
 large=C;
 }
 else
 {
 if(B>C)
 large=B;
 else
 large=C;
 }
 printf("Largest of %d, %d and %d is = %d\n", A, B, C, large);
 return 0;
}
```

**12. Write a C program (WACP) to find smallest of three numbers using nested if statement.****PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, C, small;
 printf("Enter three numbers for A , B and C\n");
 scanf("%d %d %d", &A, &B, &C);
 if(A<B)
 {
 if(A<C)
 small=A;
 else
 small=C;
 }
}
```



```
 else
 {
 if(B<C)
 small=B;
 else
 small=C;
 }
 printf("Smallest of %d, %d and %d is = %d\n", A, B, C, small);
 return 0;
}
```

**13. Write a C program (WACP) to find largest of three numbers using else...if ladder or cascaded if statement.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, C, large;
 printf("Enter three numbers for A , B and C\n");
 scanf("%d %d %d", &A, &B, &C);
 if(A>B && A>C)
 large = A;
 else if(B>A && B>C)
 large = B;
 else
 large = C;
 printf("Largest of %d, %d and %d is = %d\n", A, B, C, large);
 return 0;
}
```

**14. Write a C program (WACP) to find smallest of three numbers using else...if ladder or cascaded if statement.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, C, small;
 printf("Enter three numbers for A , B and C\n");
 scanf("%d %d %d", &A, &B, &C);
 if(A<B && A<C)
 small = A;
 else if(B<A && B<C)
 small = B;
```

```
 else
 small = C;
 printf("Smallest of %d, %d and %d is = %d\n", A, B, C, small);
 return 0;
}
```

**15. WACP to determine a given number is positive, negative or zero using nested if statement.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int number;
 printf("Enter a Number\n");
 scanf("%d", &number);
 if (number >= 0)
 {
 if(number > 0)
 printf("%d is a Postive number\n", number);
 else
 printf(" %d is ZERO\n", number);
 }
 else
 printf("%d is a Negative Number\n", number);
 return 0;
}
```

**16. WACP to determine a given number is positive, negative or zero using else...if ladder or cascaded if statement.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int number;
 printf("Enter a Number\n");
 scanf("%d", &number);
 if (number > 0)
 printf("%d is a Postive number\n", number);
 else if (number < 0)
 printf("%d is a Negative Number\n", number);
 else
 printf(" %d is ZERO\n", number);
}
```

```
 return 0;
}
```

**17. Write a C program (WACP) to find largest of three numbers using ternary operator.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, C, large;
 printf("Enter three numbers for A , B and C\n");
 scanf("%d %d %d", &A, &B, &C);
 large = (A>B) ? (A>C ? A : C) : (B>C ? B : C);
 printf("Largest of %d, %d and %d is = %d\n", A, B, C, large);
 return 0;
}
```

**18. Write a C program (WACP) to find smallest of three numbers using ternary operator.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B, C, small;
 printf("Enter three numbers for A , B and C\n");
 scanf("%d %d %d", &A, &B, &C);
 small = (A<B) ? (A<C ? A : C) : (B<C ? B : C);
 printf("Smallest of %d, %d and %d is = %d\n", A, B, C, small);
 return 0;
}
```

**19. WACP to read two numbers A and B and to print whether A is larger than, smaller than or equal to B using nested if statement.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B;
 printf("Enter two numbers for A and B \n");
 scanf("%d %d", &A, &B);
 if (A >= B)
 {
```

```
 if (A > B)
 printf("%d is Larger than %d\n", A, B);
 else
 printf("%d is Equal to %d\n", A, B);
 }
 else
 printf("%d is Smaller than %d\n", A, B);
 return 0;
}
```

**20. WACP to read two numbers A and B and to print whether A is larger than, smaller than or equal to B using else...if ladder or cascaded if statement.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B;
 printf("Enter two numbers for A and B \n");
 scanf("%d %d", &A, &B);
 if (A > B)
 printf("%d is Larger than %d\n", A, B);
 else if (A < B)
 printf("%d is Smaller than %d\n", A, B);
 else
 printf("%d is Equal to %d\n", A, B);
 return 0;
}
```

**21. WACP to read two numbers A and B and to print whether A is larger than, smaller than or equal to B using ternary operator.**

**PROGRAM:**

```
#include<stdio.h>
int main()
{
 int A, B;
 printf("Enter two numbers for A and B \n");
 scanf("%d %d", &A, &B);
 (A > B) ? printf("%d is Larger than %d\n", A, B) : (A < B ? printf("%d is Smaller than %d\n", A, B) : printf("%d is Equal to %d\n", A, B));
 return 0;
}
```

**22. WACP to find the roots of a quadratic equation using else...if ladder or cascaded if statement.****PROGRAM:**

```
#include<stdio.h>
#include<math.h>
int main()
{
 float a, b, c;
 float r1, r2, d;
 printf("Enter the coefficients a,b, and c:\n");
 scanf("%f%f%f", &a, &b, &c);
 if (a!=0 && b!=0 && c!=0)
 {
 d=(b*b)-(4*a*c);
 if (d>0)
 {
 printf("roots are real and distinct :\n");
 r1 = (-b+sqrt(d))/(2*a);
 r2 = (-b-sqrt(d))/(2*a);
 printf("root1=%f\n root2=%f\n", r1, r2);
 }
 else if (d<0)
 {
 printf ("roots are imaginary : \n");
 r1 = -b/(2*a);
 r2 = sqrt(fabs(d))/(2*a);
 printf("root1=%f + i %f\n root2=%f - i %f\n", r1, r2, r1, r2);
 }
 else
 {
 printf ("roots are real and equal: \n");
 r1 = -b/(2*a);
 r2 = r1;
 printf("root1=%f\n root2= %f\n", r1, r2);
 }
 }
 else
 printf("All coefficients must be non zero \n");
 return 0;
}
```

**23. WACP to print grade of a student based on marks obtained using else...if ladder statement.****PROGRAM:**

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

```
char grade;
printf("Enter the marks obtained by a student\n");
scanf("%d", &marks);
if (marks >= 90)
 grade='S' ;
else if (marks >= 80)
 grade='A' ;
else if (marks >= 70)
 grade='B' ;
else if (marks >= 60)
 grade='C' ;
else if (marks >= 50)
 grade='D' ;
else if (marks >= 40)
 grade='E' ;
else
 grade='F' ;
printf("Grade of a Student = %c \n", grade);
return 0;
}
```

#### **24. WACP to print grade of a student based on marks obtained using switch statement.**

##### **PROGRAM:**

```
#include<stdio.h>
int main()
{
 int marks;
 char grade;
 printf("Enter the marks obtained by a student\n");
 scanf("%d", &marks);
 marks = marks / 10;
 switch (marks)
 {
 case 10:
 case 9: grade = 'S';
 break;
 case 8: grade = 'A';
 break;
 case 7: grade = 'B';
 break;
 case 6: grade = 'C';
 break;
 case 5: grade = 'D';
 }
```

```
 break;
 case 4: grade = 'E';
 break;
 default: grade = 'F';
}
printf("Grade of a Student = %c \n", grade);
return 0;
}
```

## 25. WACP to implement a basic arithmetic calculator using else...if ladder statement.

### PROGRAM:

```
#include<stdio.h>
int main()
{
 int op1, op2;
 float result;
 char opt;
 printf("Enter the expression to be evaluated \n");
 scanf("%d %c %d", &op1, &opt, &op2);
 if (opt == '+')
 result = op1 + op2;
 else if (opt == '-')
 result = op1 - op2;
 else if (opt == '*')
 result = op1 * op2;
 else if (opt == '%')
 result = op1 % op2;
 else if (opt == '/')
 {
 if (op2 != 0)
 result = (float) op1 / op2;
 else
 {
 printf("ERROR: Divide by ZERO\n");
 return -1;
 }
 }
 else
 {
 printf("Invalid Operator \n");
 return -1;
 }
 printf("Result of %d %c %d = %f \n", op1, opt, op2, result);
}
```

```
 return 0;
 }
```

## 26. WACP to implement a basic arithmetic calculator using switch statement.

### PROGRAM:

```
#include<stdio.h>
int main()
{
 int op1, op2;
 float result;
 char opt;
 printf("Enter the expression to be evaluated \n");
 scanf("%d %c %d", &op1, &opt, &op2);
 switch (opt)
 {
 case '+': result = op1 + op2;
 break;
 case '-': result = op1 - op2;
 break;
 case '*': result = op1 * op2;
 break;
 case '%': result = op1 % op2;
 break;
 case '/': if (op2 != 0)
 {
 result = op1 / op2;
 break;
 }

 default: printf("ERROR: Divide by ZERO/Invalid Operator \n");
 return -1;
 }
 printf("Result of %d %c %d = %f \n", op1, opt, op2, result);
 return 0;
}
```

## 27. WACP to determine a given character is VOWEL or not using else...if ladder statement.

### PROGRAM:

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



```
char ch;
printf("Enter a character \n");
scanf("%c", &ch);
if (ch == 'a' || ch == 'A')
 printf("%c is Vowel \n", ch);
else if (ch == 'e' || ch == 'E')
 printf("%c is Vowel \n", ch);
else if (ch == 'i' || ch == 'I')
 printf("%c is Vowel \n", ch);
else if (ch == 'o' || ch == 'O')
 printf("%c is Vowel \n", ch);
else if (ch == 'u' || ch == 'U')
 printf("%c is Vowel \n", ch);
else
 printf("%c is not a Vowel \n", ch);
return 0;
}
```

## 28. WACP to determine a given character is VOWEL or not using switch statement.

### PROGRAM:

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

```
int main()
```

```
{
```

```
 char ch;
```

```
 printf("Enter a character \n");
```

```
 scanf("%c", &ch);
```

```
 switch (ch)
```

```
 {
```

```
 case 'a' :
```

```
 case 'A': printf("%c is a VOWEL \n",ch);
 break;
```

```
 case 'e' :
```

```
 case 'E': printf("%c is a VOWEL \n",ch);
 break;
```

```
 case 'i' :
```

```
 case 'I': printf("%c is a VOWEL \n",ch);
 break;
```

```
 case 'o' :
```

```
 case 'O': printf("%c is a VOWEL \n",ch);
 break;
```

```
 case 'u' :
```

```
 case 'U': printf("%c is a VOWEL \n",ch);
 break;
```

```

 default: printf("%c is not a VOWEL\n", ch);
 }
 return 0;
}

```

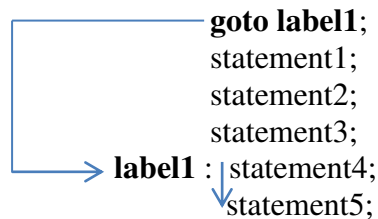
## ii. Unconditional Branching Statements

The statements that help us to jump from one statement to another statement based on condition.

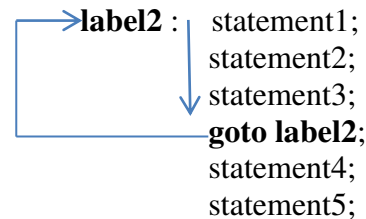
### a. goto statement

goto statement can be used to branch unconditionally from one point to another point in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by a colon (:). The label is placed immediately before the statement where the control is to be transferred.

The **syntax** is shown below:



Example 1



Example 2

In example 1, since the first statement is **goto label1** it jumps to label1 and starts executing the statement4 as shown with arrows. Hence the jump is made in forwarding direction.

In example 2, it first executes statements 1, 2 and 3 in sequence then because of the statement **goto label2**, it jumps to label2 and starts executing the statements 1, 2, and 3 and repeats the same process as shown with arrows. Hence the jump is made in backward direction and forms a loop.

From the example 2, we can understand one of the main drawback of using **goto** statement is that it forms infinite loop (i.e., loop that never ends) since it is unconditional jump statement.

Another drawback of using goto statement is that it is used to transfer the control only within the function but cannot be used to jump between functions.

**NOTE: Try to avoid using goto statement. It's a bad practice of using goto in a program.**

**Example:** Program to detect the entered number is even or odd using goto statement.

```

#include<stdio.h>
int main()
{
 int x;
 printf("Enter a Number: \n");

```

```
scanf("%d", &x);
if(x % 2 == 0)
 goto even;
else
 goto odd;
even : printf("%d is Even Number", x);
 return 0;
odd : printf("%d is Odd Number", x);
 return 0;
}
```

**Output 1:**

Enter a Number: 5  
5 is Odd Number.

**Output 2:**

Enter a Number: 8  
8 is Even Number.

**b. break statement**

**The break statement can be used in any looping statements and switch statement.**

When a break statement is encountered inside a while, for, do...while statement, then break statement immediately terminates the entire loop and jumps out of the loop and resumes execution at the next statement following the loop (if any).

When a break statement is encountered inside a switch statement case, then break statement will terminate the corresponding switch case and transfers the control out of switch statement and resumes the execution at the next statement following the switch statement (if any).

Example 1:-

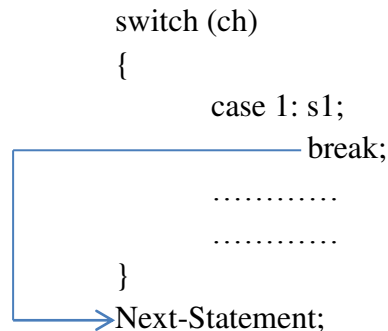
```
...
for (x = 1 ; x <= 10 ; x++)
{
 if (x > 4)
 break ;
 printf("%d ", x);
}
printf("Next Statement\n");
```

The output of the above example will be : 1 2 3 4 Next Statement.

The reason for this output is that the for loop will iterate for values of x=1, 2, 3, 4 successfully and prints 1, 2, 3, 4 on screen since these values are lesser than 4 but when value of x becomes 5 then the condition within if statement i.e.  $x > 4$  becomes

true then it executes the break statement that terminates the remaining iterations of for loop and transfers control out of loop and continues with the next statement after for loop i.e. it executes printf statement. Therefore it prints 1 2 3 4 Next Statement as output.

Example 2:-



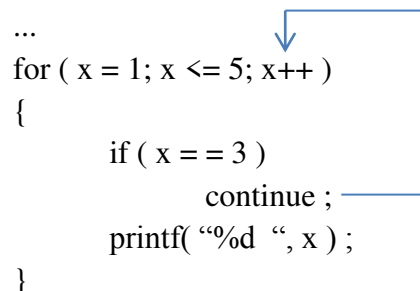
In the above example, if value of ch matches with case 1 then it executes statement s1 followed by break statement. After executing break statement the control is transferred to out of switch statement i.e. Next-Statement.

### c. continue statement

**The continue statement can be used only in looping statements.**

The continue statement terminates the current iteration of a while, for or do...while statement and resumes execution of next iteration of the loop.

For Example :-



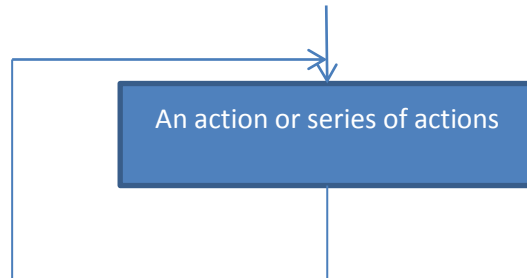
The output of the above example will be: 1 2 4 5.

The reason for this output is that the for loop will iterate for values of x=1, 2, 3, 4, and 5 but when the value of x is 3 then the condition within if statement i.e. x == 3 becomes TRUE then it executes the continue statement that terminates the current iteration of for loop (skips following printf statement) and transfers control to the beginning of next iteration of for loop i.e. x++ and continues with the remaining iterations of for loop as shown with arrow. Therefore it prints 1 2 4 5 as output.

## Looping statements:

The statements that help us to execute set of statements repeatedly are called as **looping statements**.

The concept of loop is shown in the following flowchart.



In this flowchart, the action is repeated over and over again. It never stops.

Since the loop never stops, the action or actions will be repeated forever. We don't want this to happen; we want our loop to end when the work is done. To make sure that it ends, we must have a condition that controls the loop. The condition which is used to control the loop is called as loop control expression.

Based on the placement of loop control expression, the loops are classified as TWO types

### i. Entry controlled loop or Pre-Test Loop:

In the entry controlled loop or Pre-Test loop, the loop control expression is checked before we start and at the beginning of each iteration of the loop. If the control expression is true, we execute action or actions; if the control expression is false, we terminate the loop.

Examples:

while statement and for statement

### ii. Exit controlled loop or Post-Test Loop

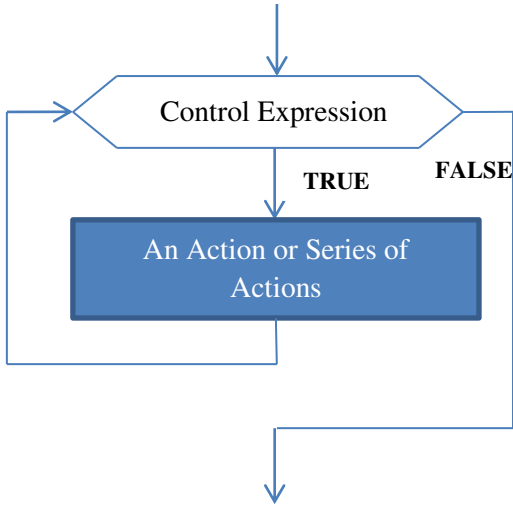
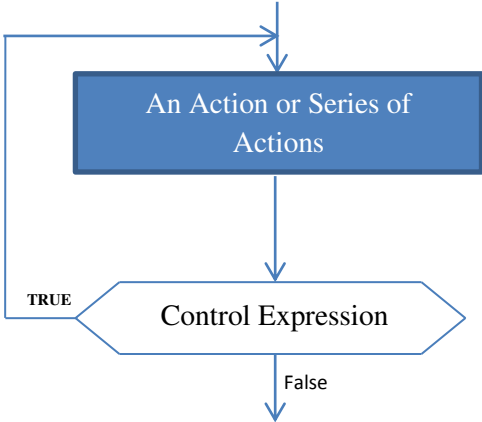
In the exit controlled loop or Post-Test loop, the loop control expression is checked after or at the end of each iteration of the loop. If the control expression is true, we repeat action or actions; if the control expression is false, we terminate the loop.

Examples:

do...while statement

The flowchart of entry controlled and exit controlled loops and other differences between the two are given below.

| Entry Controlled or Pre-Test Loop                                                                         | Exit Controlled or Post-Test Loop                                                        |
|-----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| The loop control expression is checked before we start and at the beginning of each iteration of the loop | The loop control expression is checked after or at the end of each iteration of the loop |

|                                                                                                     |                                                                                                      |
|-----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <p>Flowchart:</p>  | <p>Flowchart:</p>  |
| Minimum Iterations: Atleast ZERO Times                                                              | Minimum Iterations: Atleast ONE Times                                                                |
| If the control expression is TRUE for N times, then action or actions are executed for N times      | If the control expression is TRUE for N times, then action or actions are executed for N+1 times     |
| Examples: for loop and while loop                                                                   | Example: do...while loop                                                                             |
| Prints numbers from 1 to 10<br><pre>i=1; while( i&lt;=10) {     printf("%d\n", i);     i++; }</pre> | Prints numbers from 1 to 10<br><pre>i=1; do {     printf("%d\n", i);     i++; }while(i&lt;10);</pre> |

**Initialization:** Before a loop can start, some preparation is usually required. We call this preparation as initialization. Initialization must be done before the execution of the loop body. Initialization sets the variable which will be used in control expression to control the loop.

**Updating:** how can the control expression that controls the loop be true for a while and then change to false? The answer is that something must happen **inside** the body of the loop to change the control expression. Otherwise, the loop leads to infinite loop. Hence, the actions that cause these changes are known as **update**.

#### a. while statement

The while statement is typically used in situations where it is not known in advance how many iterations are required.

Syntax:

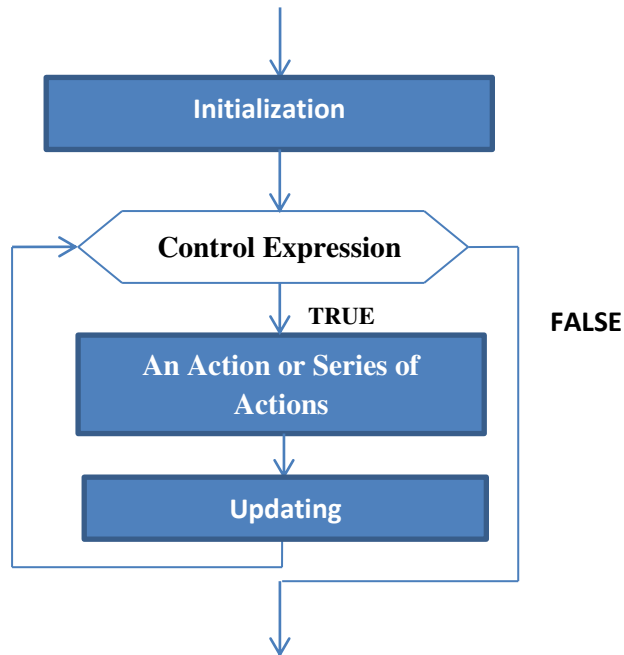
```
while (control expression)
{
 An action or Series of Actions ; //body of loop
}
```

Here, first the control expression is evaluated to TRUE or FALSE. If it is evaluated to TRUE, then an action or series of action i.e. body of loop is executed. The process of executing the body of loop is repeated until the control expression is FALSE. Once control expression is evaluated to FALSE, then the control is transferred out of the loop. The execution resumes from the statement following the loop (if any).

Generally, when initialization and updating is included in the while loop, it looks as follows.

```
Initialization // initialization is used before the start of loop and sets
 // the variable(s) to control the loop
while (control expression)
{
 An action or Series of Actions ; //body of loop
 Updating // causes the variable(s) to update that helps to repeat
 // the loop or to stop the loop
}
```

The flow of while statement is shown below.



**Example:** Program to sum all integers from 1 to 100.

```
#include <stdio.h>
int main()
{
 int sum = 0, i;
 i=1;
 while (i<=100)
 {
 sum = sum + i;
 i++;
 }
}
```

```
 printf("Sum = %d\n", sum);
 return 0;
}
```

In this example, it will repeat the statements `sum=sum+i` and `i++` till the control expression i.e. `i<=100` is **TRUE**. Once the control expression is evaluated to **FALSE**, it goes out of the loop and then executes the `printf` statement.

## b. for statement

The for statement is most often used in situations where the programmer knows in advance how many times a particular set of statements are to be repeated. The for statement is sometimes termed as counter controlled loop.

Syntax :

```
for (expression1 ; expression2 ; expression3)
{
 An action or series of actions ; // body of loop
}
```

Where,

**expression1:-** This is usually an assignment/initialization statement to set a loop control variable(s) for example. But it is not restricted to only initialization, it can be any valid C statement.

**expression2:-** This is usually a control expression which determines when loop will terminate. But, this can be any statement of C that evaluates to TRUE or FALSE.

**expression3:-** This usually defines how the loop control variable(s) will change each time the loop is executed i.e. updating.

**Body of loop:-** Can be a single statement, no statement or a block of statements.

**NOTE: All three expressions are optional. But semicolons must be present as in syntax in the loop.**

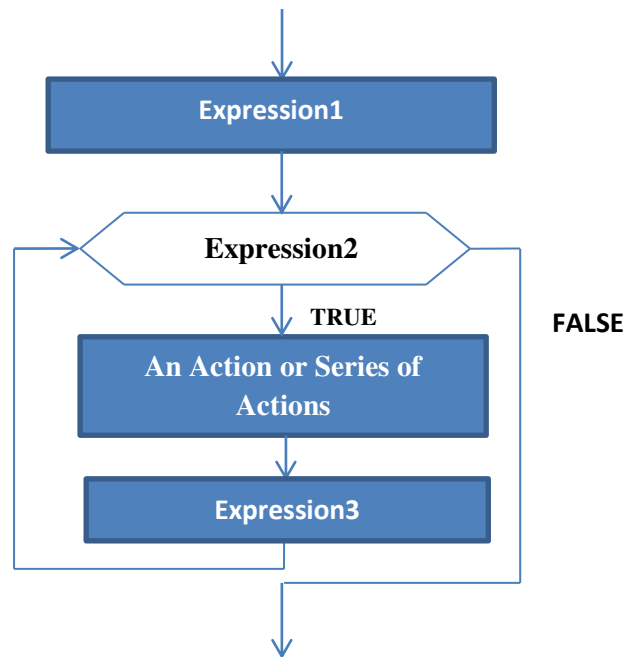
Curly braces are used in C to denote block whether in a function as in `main()` or as the body of a loop. It is optional if body of loop is a single statement or no statement.

The for statement executes as follows:

Example 1: Program to sum all numbers from 1 to 100.

```
#include <stdio.h>
int main()
{
 int x, sum=0 ;
 for (x = 1; x <= 100; x++)
 sum = sum + x;
 printf("Sum = %d\n", sum);
 return 0;
}
```





Example 2:- To print out all numbers from 1 to 100.

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

```
int main()
```

```
{
```

```
 int x;
```

```
 for (x = 1; x <= 100; x++)
```

```
 printf("%d\n", x);
```

```
 return 0;
```

```
}
```

### Multiple Initialisations in for loop

C has a special operator called the comma operator which allows separate expressions to be tied together into one statement.

For example it may be tidier to initialise two variables in a for loop as follows:-

```
for (x = 0, sum = 0; x <= 100; x++)
```

```
{
```

```
 printf("%d\n", x);
```

```
 sum += x ;
```

```
}
```

Any of the four sections associated with a for loop may be omitted but the semi-colons must be present always.

For Example 1:- expression 3 is omitted

```
for (x = 0; x < 10;)
```

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

For Example 2:- expression 1 is omitted in for loop and it is before for loop

```
x = 0 ;
```

```
for (; x < 10; x++)
```

```
printf("%d\n", x);
```

For Example 3:- An infinite loop may be created as follows

```
for (; ;)
 body of loop;
```

For Example 4:- Sometimes a for statement may not even have a body to execute as in the following example where we just want to create a time delay.

```
for (t = 0; t < big_num ; t++)
 ;
```

For Example 5:- expression 3 is a printf statement. This example prints from 1 to 100.

```
for (x = 1; x <= 100; printf("%d\n", x++)) ;
```

For Example 6:- The expression1, expression2 and expression3 sections of the for statement can contain any valid C expressions.

```
for (x = 12 * 4 ; x < 34 / 2 * 47 ; x += 10)
 printf("%d ", x) ;
```

Nested for loops:- It is possible to build a nested structure of for loops, for example the following creates a large time delay using just integer variables.

```
unsigned int x, y ;
for (x = 0; x < m; x++)
 for (y = 0; y < n; y++)
 ;
```

In this example, the outer for loop will iterate for 'm' times and inner loop will iterate for 'n' times, hence the total iterations from nested loop is **m\*n times**.

For Example: Program to produce the following table of values

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 4 | 5 | 6 |
| 3 | 4 | 5 | 6 | 7 |
| 4 | 5 | 6 | 7 | 8 |
| 5 | 6 | 7 | 8 | 9 |

```
#include <stdio.h>
int main()
{
 int j, k ;
 for (j = 1; j <= 5; j++)
 {
 for (k = j ; k < j + 5; k++)
 {
 printf("%d ", k) ;
 }
 printf("\n") ;
 }
 return 0;
}
```

**c. do...while statement**

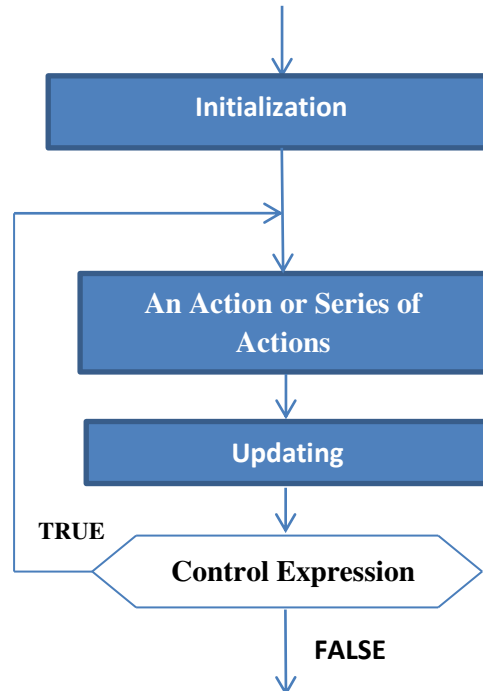
The terminating condition in the for and while loops is always tested before the body of the loop is executed -- so of course the body of the loop may not be executed at all.

In the do...while statement on the other hand the body of the loop is always executed at least once as the condition is tested at the end of the body of the loop.

Syntax :

```
Initialization
do
{
 An action or Series of Actions ; // body of loop
 Updating
} while (control expression) ;
```

The flow of do...while loop is shown below



For Example : To sum all numbers from 1 to 100.

```
int i, sum=0;
i = 1;
do
{
 sum = sum + i ;
 i++;
} while (i <= 100) ;
```

**Programming examples on Looping:****1. WACP to find the sum of numbers from 1 to N where N is entered from keyboard.****Program:**

**// sum of numbers from 1 to N i.e.  $sum = 1+2+3+4+5+...+N$**

```
#include<stdio.h>
int main()
{
 int N, sum=0, i;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 i=1;
 while(i<=N)
 {
 sum = sum + i;
 i++;
 }
 printf("Sum = %d\n", sum);
 return 0;
}
```

**2. WACP to find the sum squares of numbers from 1 to N where N is entered from keyboard.****Program:**

**// sum of squares of numbers from 1 to N i.e.  $sum = 1^2+2^2+3^2+4^2+5^2+...+N^2$**

```
#include<stdio.h>
#include<math.h>
int main()
{
 int N, sum=0, i;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 i=1;
 while(i<=N)
 {
 sum = sum + pow(i,2);
 i++;
 }
 printf("Sum = %d\n", sum);
 return 0;
}
```

**3. WACP to evaluate the following  $f(x) = 1 + X + X^2 + X^3 + \dots + X^N$ , where X and N are entered from keyboard.**

**Program:**

```
// f(x) = 1 + X + X^2 + X^3 + ... + X^N
#include<stdio.h>
#include<math.h>
int main()
{
 int N, X, sum=0, i;
 printf("Enter the value of X and N\n");
 scanf("%d%d", &X, &N);
 for(i=0; i<=N; i++)
 sum = sum + pow(X,i);
 printf("Sum = %d\n", sum);
 return 0;
}
```

**4. WACP to find and print reverse number of a given integer number.**

**Program:**

```
#include<stdio.h>
int main()
{
 int N, reverse=0, rem, temp;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 temp=N;
 while(N>0)
 {
 rem = N%10;
 reverse = reverse * 10 + rem;
 N = N/10;
 }
 printf("Given Number = %d\n", temp);
 printf("Reversed Number = %d\n", reverse);
 return 0;
}
```

**5. WACP to find reverse number of a given integer number and print whether it is a palindrome or not.**

**Program:**

```
#include<stdio.h>
int main()
{
 int N, reverse=0, rem, temp;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 temp = N;
 while(N>0)
 {
 rem = N%10;
 reverse = reverse * 10 + rem;
 N = N/10;
 }
 printf("Given Number = %d\n", N);
 printf("Reversed Number = %d\n", reverse);
 if(N == temp)
 printf("%d is Palindrome\n", temp);
 else
 printf("%d is not Palindrome\n", temp);
 return 0;
}
```

**6. WACP to find sum of digits of a given integer number. ( For example, 123 = 1 + 2 + 3 = 6).**

**Program:**

```
#include<stdio.h>
int main()
{
 int N, sum=0, rem;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 while (N>0)
 {
 rem = N%10;
 sum = sum + rem;
 N = N/10;
 }
 printf(" Sum of Digits = %d\n", sum);
 return 0;
}
```

```
}
```

## 7. WACP to find the factorial of a given number.

### Program:

```
#include<stdio.h>
int main()
{
 int N, fact=1, i;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 i=1;
 do
 {
 fact = fact * i;
 i++;
 } while(i<=N);
 printf("Factorial(%d) = %d\n", N, fact);
 return 0;
}
```

## 8. WACP to find the GCD and LCM of given integer numbers.

### Program:

```
#include<stdio.h>
int main()
{
 int M, N, P,Q, GCD, rem, LCM;
 printf("Enter the value of M and N\n");
 scanf("%d%d", &M, &N);
 P=M;
 Q=N;
 while (N>0)
 {
 rem = M%N;
 M=N;
 N = rem;
 }
 GCD=M;
 printf(" GCD (%d, %d) = %d\n", P, Q, GCD);
 LCM = (P* Q) / GCD;
 printf(" LCM (%d, %d) = %d\n", P, Q, LCM);
 return 0;
}
```

**9. WACP to find the sum of even and odd numbers from 1 to N.****Program:**

```
#include<stdio.h>
int main()
{
 int N, i, even_sum=0, odd_sum=0;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 for(i=1; i<=N; i++)
 {
 if(i % 2 == 0)
 even_sum += i;
 else
 odd_sum += i;
 }
 printf(" Sum of Even Numbers = %d\n", even_sum);
 printf(" Sum of Odd Numbers = %d\n", odd_sum);
 return 0;
}
```

**10. WACP to print alternate uppercase letters starting with letter 'A'.****Program:**

```
#include<stdio.h>
int main()
{
 char ch;
 printf("Alternate Uppercase letters starting with letter A are:\n");
 for(ch='A'; ch<='Z'; ch= ch + 2)
 printf("%c\n", ch);
 return 0;
}
```

**11. WACP to find the GCD of two numbers using while and ternary operator.****Program:**

```
#include<stdio.h>
int main()
{
 int M, N, GCD, P,Q;
 printf("Enter the value of M and N\n");
 scanf("%d%d", &M, &N);
 P=M;
```



```
 Q=N;
 while (M != N)
 (M>N) ? M = M - N : N = N - M ;
 GCD=M;
 printf(" GCD (%d, %d) = %d\n", P,Q, GCD);
 return 0;
}
```

**12. WACP to find the GCD of two numbers using while and if else statement.**

**Program:**

```
#include<stdio.h>
int main()
{
 int M, N, GCD, P,Q;
 printf("Enter the value of M and N\n");
 scanf("%d%d", &M, &N);
 P=M;
 Q=N;
 while (M != N)
 {
 if (M>N)
 M = M - N;
 else
 N = N - M ;
 }
 GCD=M;
 printf(" GCD (%d, %d) = %d\n", P, Q, GCD);
 return 0;
}
```

**13. WACP to find a given number is prime or not.**

**Prime number:** A number which is divisible by 1 and itself.

**Program:**

```
#include<stdio.h>
int main()
{
 int N, i;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 for(i=2; i<=N/2; i++)
 {
 if(N % i == 0)
```

```
 {
 printf("%d is not a prime number\n", N);
 return 0;
 }
 }
 printf("%d is a prime number\n", N);
 return 0;
}
```

\*\*\*\*\*ALL THE BEST\*\*\*\*\*

## MODULE 3: Arrays, Functions and Strings

### Contents covered in this module

- I. Using an Array
- II. Functions in C
- III. Argument Passing
- IV. Functions and Program Structure, locations of functions
- V. Function Design
- VI. Recursion
- VII. Multidimensional Arrays
- VIII. Using Arrays with Functions
- IX. Declaring, Initializing, printing and reading strings
- X. String Input and Output Functions
- XI. String Manipulation Functions
- XII. Array of Strings
- XIII. Programming Examples

### I. Using an Array

Array is collection of similar data items or elements. Array is a collection of elements of the same data type. All the elements of the array are stored in contiguous (sequential) memory locations.

**Array is a sequential collection of elements of same type under a single name.**

Pictorial representation of an array is

|                                |                         |      |      |             |      |
|--------------------------------|-------------------------|------|------|-------------|------|
| Array Element                  | 10                      | 13   | 4    | 26          | 38   |
| Index or Position or Subscript | 0                       | 1    | 2    | 3           | 4    |
| Address (Assumed Values)       | 1000 (starting address) | 1004 | 1008 | <b>1012</b> | 1016 |

In the above representation, an array is a collection of 5 integer elements.

The addresses of each element of an array are contiguous in nature. Hence, the addresses are 1000, 1004, 1008, 1012 and 1016 since **size of each integer data is 4 bytes**.

Address of any element of an array can be calculated by using the following formula

$$\text{Element Address} = \text{Starting Address} + (\text{sizeof(element)} * \text{index})$$

For instance, 4<sup>th</sup> element index is 3 from the above, hence

$$\begin{aligned} 4^{\text{th}} \text{ element address} &= 1000 + (4 * 3) \\ &= 1000 + 12 \\ &= \mathbf{1012} \end{aligned}$$

The position or index of first element of an array is always ZERO (0). The index or position of next elements is one more than the previous index or position. Hence, the indices in the representations are 0, 1, 2, 3, and 4.

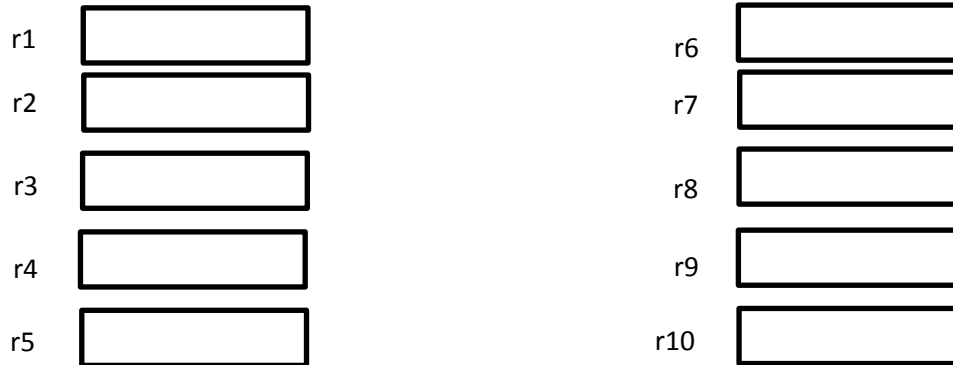
### Why Arrays? or What is the need for arrays?

Imagine we have a problem that requires us to read, process and print 10 integers. To begin, we can declare and define 10 variables, each with a different name as shown below;

**int r1, r2, r3, r4, r5, r6, r7, r8, r9, r10;**

This creates 10 variables and allocates 10 different blocks of memory one for each variable as shown below.

Having 10 different names, how can we read 10 integers from the keyboard and store them? To read 10 integers from the keyboard, we need 10 read statements, each to a different variable. Furthermore, once we have them in memory, how can we print them? To print them we need 10 print statements.



For example, the following program illustrates how to read and print 10 integer numbers into 10 different names.

```
#include<stdio.h>
int main()
{
 int r1, r2, r3, r4, r5, r6, r7, r8, r9, r10;
 // read 10 integers into 10 different names
 scanf("%d", &r1);
 scanf("%d", &r2);
 scanf("%d", &r3);
 scanf("%d", &r4);
 scanf("%d", &r5);
 scanf("%d", &r6);
 scanf("%d", &r7);
 scanf("%d", &r8);
 scanf("%d", &r9);
 scanf("%d", &r10);

 //print 10 integer numbers from 10 different names
 printf("%d", r1);
 printf("%d", r2);
 printf("%d", r3);
 printf("%d", r4);
 printf("%d", r5);
 printf("%d", r6);
 printf("%d", r7);
```

```
 printf("%d", r8);
 printf("%d", r9);
 printf("%d", r10);

 return 0;
 }
```

Although this approach may be acceptable for 10 variables but how about 100 variables, 1000 variables or 10000 variables? It is definitely not acceptable.

Hence, we need more powerful data structure to process large amounts of data. Hence the array is used for processing large amounts of data of same type.

By using arrays the above program can be rewritten as follows

```
#include<stdio.h>
int main()
{
 int r[10];
 int i;
 // read 10 integer numbers
 for(i=0;i<10;i++)
 scanf("%d", &a[i]);
 //print 10 integer numbers
 for(i=0;i<10;i++)
 printf("%d", a[i]);
 return 0;
}
```

This example program is much easier to read, understand and alter the number of elements to be read and print as per the requirement. Hence, the arrays are much better than variables for storing large amounts of same type of data. Hence the various applications and advantages are listed below.

### Applications of Arrays

Various applications and advantages of arrays are

- Can store large elements of same type
- Can replace multiple different variable names of same type by single name
- Can be used for sorting and searching applications
- Can be used for representing matrix
- Can be used in recursion
- Can be used for implementing various data structures like stack, queue, etc.

### Different Types of Arrays

In C, the arrays can be classified as two types based on the arrangement of data.

- a. One-Dimensional Array
- b. Multi-Dimensional Array

#### One-Dimensional Array:

An array is one in which elements are arranged linearly. If an array contains only one subscript then it is called as one-dimensional array.

## What is Index or Subscript?

- Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript / index
- Subscripts helps us to identify the element number to be accessed in the contiguous memory

## Declaration and definition of one-dimensional array

An array must be declared and defined before it can be used. Array declaration and definition tells the compiler

- Name of the array
- The size or the number of elements in the array
- The type of each element of the array
- Total amount of memory to be allocated to the array

**The size of the array is a constant and must have a value at compilation time.**

The declaration format or syntax of declaration is given below;

**Type                      array\_name [arraysize] ;**

Where,

**Type** - refers to the type of each element of the arrays which can be int, float, char, double, short, long, etc.

**array\_name** - refers to the name of the array to be specified with the help of rules of identifier

**arraysize** - refers to the size or number of elements of the array. This **arraysize** can be a symbolic constant, integer constant or integer arithmetic expression.

**Examples:**

- a. declare and define an array by the name **rollno** which contains **50 integer** elements. In this example, **arraysize** is an **integer constant**.

int     rollno[50];

- b. declare and define an array called **average** which contains 10 floating point elements. In this example, **arraysize** is an **integer constant**.

float   average[10];

- c. #define     MAX 1000     //defines a macro MAX with value 1000

char   string [MAX];     // declares and defines an array by the name **string** that can contain **1000 characters**. In this example, **arraysize** is **symbolic constant**.

- d. double     salary[10+5];     // declares and defines an array called **salary** which can contain 15 (result of 10 + 5) double precision floating point numbers. In this example, **arraysize** is **integer arithmetic expression**.

## Accessing Elements in Arrays

C uses an **index** to access individual elements in an array. The index must be an integral value or an expression that evaluates to an integral value.

For example, consider the following array

**int     marks[5];**

To access **first element** of the array **marks**, we can use **marks[0]**, since the index of first element is ZERO (0).

Similarly, second element of the array can be accessed by marks[1], third element by marks[2], etc.

### Storing Values in Arrays

Declaration and definition only reserve space for the elements in the array. No values are stored.

If we want to store values in the array, we can use the following methods of initialization

- initialize the elements (**Initialization**)
- read values from the keyboard (**Inputting**)
- assign values to each individual element (**Assignment**)

### Initialization of an array

Providing a value for each element of the array is called as initialization. The list of values must be enclosed in curly braces.

**NOTE: It is a compile time error if we provide more values than the size of the array.**

Examples:

a) basic initialization

```
int numbers[5] = { 3, 23, 17, 8, 19};
```

In this example, first element numbers[0] is initialized to 3  
second element numbers[1] is initialized to 23  
third element numbers[2] is initialized to 17  
fourth element numbers[3] is initialized to 8  
fifth element numbers[4] is initialized to 19

b) initialization without size of the array

```
int numbers[] = { 3, 7, 12, 29, 30 };
```

In this example, since the size of the array is not specified in brackets, size of array is decided based on number of elements in the curly braces. Hence the size of array is 5.

c) Partial initialization

```
int numbers[5] = {4, 9};
```

In this example, the array is partially initialized, since the number of values provided in curly braces are fewer than the size of the array. Therefore, it initializes first element with 4, second element with 9 and unassigned or uninitialized elements are filled with ZEROs automatically.

d) Initialization of all elements to ZEROs

```
int numbers[100]= {0};
```

In this example, all the elements are filled with ZEROs.

### Inputting Values

Another way to fill the array is to read the values from the keyboard or file by using scanf statement or fscanf statement.

For Example,

```
int numbers[5];
int i;
for(i=0;i<5;i++)
 scanf("%d", &numbers[i]);
```

In this example, scanf function is repetitively called for 5 times and reads the values for all elements of the array. Since the starting index of array is ZERO, so index **i** is initialized to 0 in for loop and the condition  $i < 5$  iterates loop for 5 times i.e., the number of elements in the array.

### Assigning values

We can assign values to individual elements using the assignment operator. Any value that reduces to the proper type of array can be assigned to an individual array element.

For Example,

```
int numbers[5];
numbers[0] = 10; //assigns first element to 10
numbers[1] = 4; //assigns second element to 4
```

## II. Functions in C

Function is an independent module which contains set of instructions to perform a particular task.

### Why Functions? Or What is the need for functions in C?

Since C is modular programming language, the larger problem or complex problem can be subdivided into smaller problems, each of these smaller problems can be developed independently by using functions. Hence, by using functions in C, we can have so many advantages listed below. The process of subdividing larger problem into smaller problem is called modular approach.

Advantages of Functions

- It is easy to read, understand, test, manage, and debug the programs.
- Reduces the development time
- Reusability of code can be done
- We can protect data, since one function cannot access local data of another function.
- Complexity of the program can be reduced.
- Users can create their own libraries i.e user-defined libraries.

### Types of Functions in C

There are two types of functions in C.

#### a. Standard functions or library functions:

The functions which have pre-defined purpose and readily available for use in header files are called as standard functions or library functions.

Examples:

|          |                                                                                |
|----------|--------------------------------------------------------------------------------|
| sqrt()   | - used to find square root of a given number. Available in math.h header file. |
| sin()    | - used to find sin(x). Available in math.h header file.                        |
| printf() | - used to display data in required form. Available in stdio.h header file.     |
| scanf()  | - used to read data in required form. Available in stdio.h header file.        |

#### b. User-defined functions:



The functions which are defined by user to perform a specific task are called as user-defined functions.

### Using User-defined functions in C

To understand how to use user-defined functions in C and how program is executed when functions are used in C, we will start with a simple program and its flow of execution. One must understand different elements of functions and various statements to be used while designing functions in C program.

#### Explanation:

The below shown program illustrates finding a square of a given number using a user-defined function `square()`. It also illustrates flow of program execution and the three elements of function.

Understanding flow of program execution

- a. Firstly Operating System will call main function.
- b. When control comes inside main function, execution of main starts (i.e execution of C program starts) and executes Line 5 first
- c. Consider Line 6 i.e., `result = square(5)`,

In this statement, a function call is made to the user-defined function **`square()`** which accepts one parameter i.e., 5 to find a square of 5. Hence, it halts or stops the execution of main function at line 6 and the control is transferred to the function definition i.e., line 10 as shown with arrow. The value of function parameter is copied to variable `x`.

- d. Then function definition is executed line by line i.e., from line 10 to line 14.
- e. Consider line 14 i.e. `return y`,

In this statement the value of `y` i.e. 25 is returned to the calling function i.e. main function. Hence, the called function i.e., `square()` completes its execution and returns the control back to calling function i.e. main function to the line 6 as shown with arrow. The return value i.e. 25 is copied to variable `result`.

- f. Then main function will resume its execution and executes its statements.
- g. Lastly the main function terminates with return value 0 to the operating system.

From the example, we can also define the following basic definitions related to functions of C.

**Calling Function:** The function which invokes another function to do something is known as calling function.

**Called Function:** The function which is invoked by another function is known as called function.

**Actual Parameters:** the parameters in a function call which are used to send one or more values to the called function are known as actual parameters.

For example:

If a function call is

`addition(p,q);`

Then in this function call the parameters listed in parenthesis namely variables `p` and `q` are actual parameters whose value is passed to the formal parameters.

**Formal Parameters:** the parameters in function declaration or definition which are used to receive one or more values from the calling function are known as formal parameters.

For example:

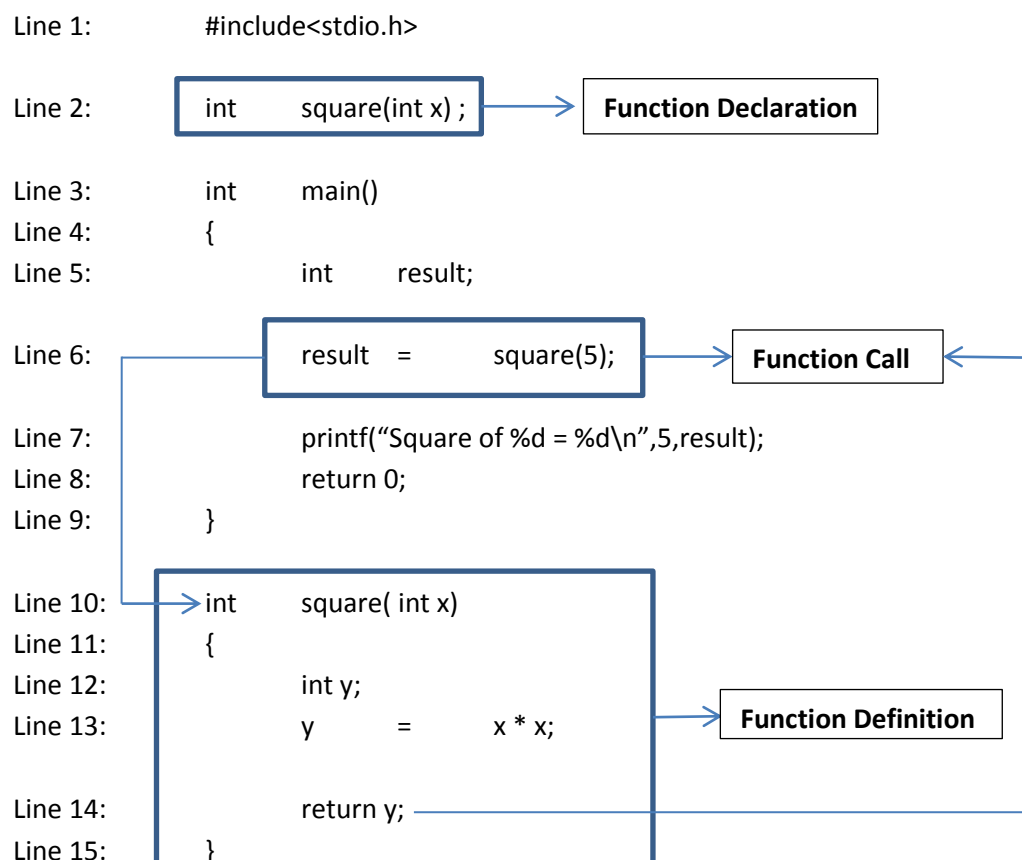
If a function definition is

```
void addition(int x, int y)
{
 printf("sum = %d\n", x+y);
}
```

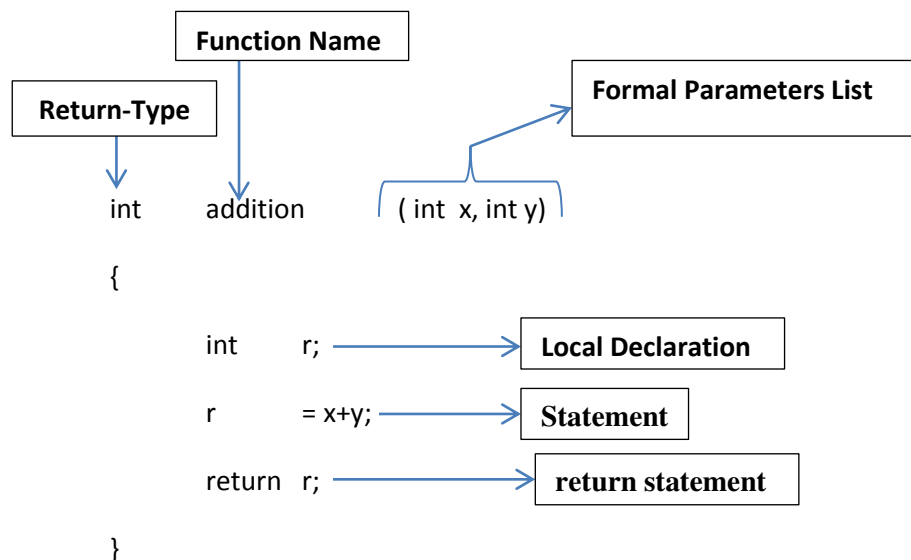
In this function definition, the parameters listed in function header namely x and y are formal parameters which will receive the copy of actual parameters.

**Return statement:** the statement which is used to return a value to the calling function from called function is known as return statement.

**Function Name:** the name given to the function which must be framed with the help of rules of identifier.



Pictorial view of all above definitions can be shown as given below



## Elements of Function

From the above example, one can observe that to define user-defined functions in C program, the three elements of functions are very important and need to understand how to use and purpose of these elements.

Three elements of function are

1. Function Declaration or Function Prototype
2. Function call
3. Function Definition

### 1. Function Declaration or Function Prototype or Function Header

As we declare variables, arrays, etc, the function must be declared before we use it. The function declaration tells the compiler that we are going to define this function somewhere in the program. It gives prior information about the function to the compiler. It contains only a function header.

The declaration tells the compiler about

- Name of the function
- Return type of the function
- List of formal parameters and their types

Hence, the general format or syntax for declaring function is as given below. The declaration must end with semicolon.

```
return-type Function_Name (List of formal parameters) ;
```

Where,

**return-type** can be

- void type when function doesn't return any value
- any of the standard data type or user-defined data type when function returns a value

for example: int, float, double, char, short int, long int, etc

**Function\_Name** can be any name framed by using the rules of identifier

**List of formal parameters:** refers to zero or more parameters to receive either value or address of actual parameters.

Format for writing list of formal parameters

Type1 param1, Type2 param2, Type3 param3, ..., TypeN paramN

Where,

Type1, Type2, ..., TypeN can be any valid data types of C

Param1, param2, ..., paramN are names of formal parameters. All names must be unique. The names must be framed by using rules of identifier.

**For example:**

a.     int     addition(int x, int y);     // here the function name is addition which accepts two parameters of type integer and returns an integer value.

b.     void    square(float a);             // here function name is square which accepts one parameter of floating point type and doesn't return any value.

c.     char    str(char c, int a);          // here function name is str which accepts two parameters one of type character and another of integer type and returns a character.

## 2. Function Call

Function call is an operator which is used to invoke a function with or without parameters or arguments known as actual parameters. The actual parameters identify the values or addresses that are to be sent to the called function. They can contain zero or more parameters. They must match the functions formal parameters type and order in the parameter list. Function call statement must end with semicolon.

General Syntax or format of function call is

variable = Fuction\_Name (list of actual parameters) ;

Where,

Variable is one which receives the return value of function if the function return type is Non-Void. If the function return type is void then variable must be ignored.

Function\_Name must match to name of the function which is declared before.

List of actual parameters can be zero or more names of variables to send either value or address.

For example,

a.     sum = addition(p, q);             // it invokes function addition with two actual parameters p and q and return value of function is stored into variable sum.

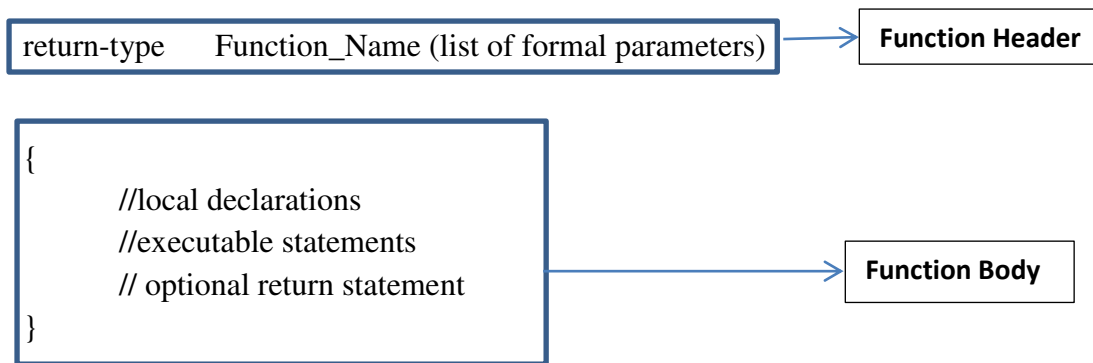
b.     square(x);     // invokes a function square with only one actual parameter x

c.     ch = str(ch1, x);                 // invokes a function str with two parameters ch1 and x and stores the return value in variable ch.

## 3. Function Definition

The function definition contains the code for a function. It is made up of two parts: the function header and the function body, which is a block of statements enclosed in pair of curly braces.

Syntax of function definition is



A function header consists of three parts: return-type, the function name and list of formal parameters. A semicolon is not used at the end of the function header.

Function body contains local declarations and the set of executable statements to perform a particular task.

If the function return-type is void then function definition must not contain return statement or it can have empty return statement. If the function return-type is non-void then function definition must return with appropriate type of value using return statement.

For example,

```
a. int addition(int x, int y)
 {
 int sum;
 sum = x+y;
 return sum;
 }
```

In this function definition, x and y are formal parameters, sum is a local variable. The function returns value of sum using return statement.

```
b. void square(float x)
 {
 printf("square of %f = %f\n", x, x*x);
 }
```

In this function definition, x is a formal parameter of type float. This function prints square of x. Function doesn't return any value since return type is void hence, it has no return statement.

### III. Parameters Passing Mechanisms or Argument Passing Mechanisms

In C, we can two types of communications such as one-way communication and two-way communication. One-way communication may occur from calling function to called function or from called function to calling function. Two-way communication is also known as bidirectional communication which occurs between both calling and called function.

The strategies used to implement inter-function communication between the functions are known as parameter passing mechanisms. There are three mechanisms to implement inter-function communications.

1. Call-by-Value or Pass-by-Value method
2. Call-by-Reference or Pass-by-Reference (call-by-address or pass-by-address)
3. Return statement

### 1. Call-by-Value or Pass-by-Value method

In this mechanism, the values of actual parameters in a function call are copied to the formal parameters.

Any modification to the formal parameters in the function definition will not affect actual parameters.

Default method of function call is call-by-value mechanism.

#### Example: Program to Swap two integer numbers

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

```
void swap(int x, int y);
```

```
int main()
```

```
{
 int p, q;
 printf("Enter the value for p and q\n");
 scanf("%d%d", &p, &q);
 printf("In Main: Before Function Call\n");
 printf("p=%d, q=%d\n", p, q);
```

```
 swap(p,q);
```

```
 printf("In Main: After Function Call\n");
 printf("p=%d, q=%d\n", p, q);
 return 0;
```

```
}
```

```
void swap(int x, int y)
```

```
{
 int temp;
 printf("In Swap: Before Exchange\n");
 printf("x=%d, y=%d\n", x, y);
 temp=x;
 x=y;
 y=temp;
 printf("In Swap: After Exchange\n");
 printf("x=%d, y=%d\n", x, y);
}
```

Copies value of p=10 to formal parameter x and value of q=20 to formal parameter y and transfers control to function definition

#### Output of the above program is

Enter the value for p and q

10    20

**In Main: Before Function Call**

**p=10, q=20**

In Swap: Before Exchange

x=10, y=20

In Swap: After Exchange

x=20, y=10

### In Main: After Function Call

**p=10, q=20**

From the above output and flow shown with arrow and text description, we can understand that the values of actual parameters p and q are copied to formal parameters x and y respectively. From the bold text of output we can understand that the modification that we have done for formal parameters in function definition have not affected actual parameters in calling function hence, the p and q value before function call and after function is same.

## 2. Call-by-reference or Pass-by-reference method

In this mechanism, the addresses of actual parameters in a function call are copied to the formal parameters.

Any modification to the formal parameters in the function definition will affect actual parameters.

### Example: Program to Swap two integer numbers

// Pointer is a variable which contains address of another variable

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

```
void swap(int *x, int *y);
```

```
int main()
```

```
{
```

```
 int p, q;
```

```
 printf("Enter the value for p and q\n");
```

```
 scanf("%d%d", &p, &q);
```

```
 printf("In Main: Before Function Call\n");
```

```
 printf("p=%d, q=%d\n", p, q);
```

```
 swap(1000,2500)
```

```
 swap(&p,&q);
```

```
 printf("In Main: After Function Call\n");
```

```
 printf("p=%d, q=%d\n", p, q);
```

```
 return 0;
```

```
}
```

```
void swap(int *x, int *y)←
```

```
{
```

```
 int temp;
```

```
 printf("In Swap: Before Exchange\n");
```

```
 printf("x=%d, y=%d\n", *x, *y);
```

```
 temp=*x;
```

```
 *x=*y;
```

```
 *y=temp;
```

```
 printf("In Swap: After Exchange\n");
```

```
 printf("x=%d, y=%d\n", *x, *y);
```

```
}
```

Copies address of p (&p) assumed that it is at address 1000 to formal parameter \*x and address of q (&q) assumed that it is at address 2500 to formal parameter \*y and transfers control to function definition

**Output of the above program is**

Enter the value for p and q

10     20

**In Main: Before Function Call**

**p=10, q=20**

In Swap: Before Exchange

x=10, y=20

In Swap: After Exchange

x=20, y=10

**In Main: After Function Call**

**p=20, q=10**

From the above output and flow shown with arrow and text description, we can understand that the address of actual parameters p and q are copied to formal parameters \*x and \*y respectively. From the bold text of output we can understand that the modification that we have done for formal parameters in function definition have affected(modified) actual parameters in calling function hence, the p and q value before function call and after function are different.

#### IV. Functions, Program Structure and Location of functions

When functions are used with C then general structure of the C program can be organized in two different ways based on **placement or location of the function definition**. C program can be designed in two different approaches namely; top-down approach and bottom-up approach. Hence the program with functions can be organized in two-different ways.

In **top-down approach**, main function is designed and developed first then sub functions are designed and developed hence it is known as top-down approach. The program runs from top to bottom.

In **bottom-up approach**, the sub functions are designed and developed first then main function is designed and developed hence the name bottom-up. The program runs from bottom to up.

##### General Structure for Top-Down approach

Preprocessor Directives

Global Declarations

User-Defined Functions Declarations

int main()

{

    Local Declarations

    Executable Statements

    Functions call

}

User-Defined Function Definitions

##### General Structure for Bottom-Up approach

Preprocessor Directives

Global Declarations

User-Defined Functions definitions



```
int main()
{
 Local Declarations
 Executable Statements
 Functions call
}
```

### **Example for Top-Down Approach**

**//program to print a greeting message using top-down approach**

```
#include<stdio.>
void greeting(void);
int main()
{
 greeting();
 return 0;
}
void greeting(void)
{
 printf("Good Morning!!!Have a Good Day\n");
}
```

Here, in this program main function is designed first which invokes sub function greeting() to display a greeting message. So sub function is designed later. The program starts its execution from main function and flows through sub function greeting hence, it is top-down approach.

### **Example for Bottom-Up Approach**

**//program to print a greeting message using bottom-up approach**

```
#include<stdio.>
void greeting(void)
{
 printf("Good Morning!!!Have a Good Day\n");
}
int main()
{
 greeting();
 return 0;
}
```

Here, in this program sub function greeting is designed and developed first then main function is designed which invokes sub function greeting() to display a greeting message. The program starts its execution from main function (bottom) then flows through sub function greeting() (which is above main()) hence, it is bottom-up approach.

## **V.Function Design**

The functions can be designed in four different forms to establish the communications such as one-way communication or two-way communication. These function designs are

categorized based on return value of the function and parameters accepted by the function. So they are also called as categories of function designs.

The four different forms or categories of function design are:

1. Void function without parameters
2. Void function with parameters
3. Non-Void function without parameters
4. Non-Void function with parameters

### 1. Void function without parameters

In this design, the function doesn't return any value and doesn't accept any parameters so it is also called as function with no return value and no parameters.

When function doesn't return value then return type must be void.

Consider the following example program which performs addition of two integer numbers with the help of function.

```
#include<stdio.h>
void addition(void);
int main()
{
 addition();
 return 0;
}
void addition(void)
{
 int p, q, sum;
 printf("Enter p and q\n");
 scanf("%d%d", &p, &q);
 sum = p + q;
 printf("sum = %d\n", sum);
}
```

In this program, the sub function addition() is not returning any value and not even taking any parameters. The full control is given to sub function which reads two integer numbers, performs addition and then prints the sum.

### 2. Void function with parameters

In this design, the function doesn't return any value but accepts one or more parameters so it is also called as function with no return value but with parameters.

When function doesn't return value then return type must be void.

Consider the following example program which performs addition of two integer numbers with the help of function.

```
#include<stdio.h>
void addition(int x, int y);
int main()
{
 int a, b;
 printf("Enter a and b\n");
 scanf("%d%d", &a, &b);
}
```

```
 addition(a,b);
 return 0;
 }
void addition(int x, int y)
{
 int sum;
 sum = x + y;
 printf("sum = %d\n", sum);
}
```

In this program, the sub function addition() is not returning any value but taking two parameters x and y which receives copy of a and b respectively. It is a type of down-ward one-way communication since, main function i.e calling function is sending data to sub function addition i.e. called function.

### 3. Non-Void function without parameters

In this design, the function returns a value but doesn't accept any parameters so it is also called as function with return value and no parameters.

When function returns a value then return type must be non-void.

Consider the following example program which performs addition of two integer numbers with the help of function.

```
#include<stdio.h>
int addition(void);
int main()
{
 int sum;
 sum = addition();
 printf("sum = %d\n", sum);
 return 0;
}
int addition(void)
{
 int sum, a, b;
 printf("Enter a and b\n");
 scanf("%d%d", &a, &b);
 sum = a + b;
 return sum;
}
```

In this program, the sub function addition() is returning an integer value i.e. sum but not taking any parameters. It is a type of up-ward one-way communication since sub function i.e called function is sending data to main function i.e. calling function.

### 4. Non-Void function with parameters

In this design, the function returns a value and also accepts one or more parameters so it is also called as function with return value and with parameters.

When function returns a value then return type must be non-void.

Consider the following example program which performs addition of two integer numbers with the help of function.

```
#include<stdio.h>
int addition(int x, int y);
int main()
{
 int a, b, sum;
 printf("Enter a and b\n");
 scanf("%d%d", &a, &b);
 sum = addition(a,b);
 printf("sum = %d\n", sum);
 return 0;
}
int addition(int x, int y)
{
 int sum;
 sum = x + y;
 return sum;
}
```

In this program, the sub function addition() is returning an integer value i.e. sum and also taking two parameters x and y which receives copy of a and b respectively. It is a type of two-way communication since, main function i.e calling function is sending data to sub function addition i.e. called function as well as called function is also returning a value to calling function.

## VI. Recursion

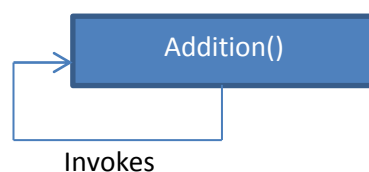
Recursion is a process in which a function calls itself. This enables the function to call several times or repeatedly to solve the given problem. The function which is called recursively by itself is known as recursive function. Since recursion initiates repetition of same function there must exist a stopping condition.

The various types of recursion are

- a. Direct recursion
- b. Indirect recursion
- a. **Direct recursion**

A function that invokes itself is said to be direct recursion.

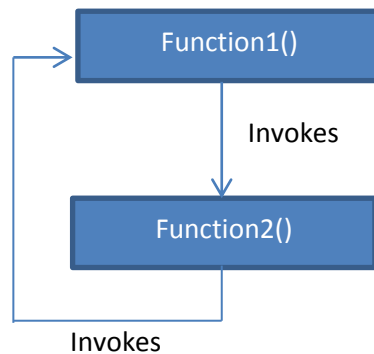
The following picture depicts the direct recursion



In this picture, a function known as Addition() is invoking Addition() recursively i.e., a function is called by itself hence direct recursion.

**b. Indirect recursion**

A function is recursively invoked by another function then it is known as indirect recursion.



In this picture, it illustrates that Function1 is invoking Function2 in turn Function2 is invoking Function1 hence it forms recursion. Here, Function1 is called recursively by Function2 and even Function2 is called recursively by Function1 hence, both Function1 and Function2 are indirect recursive functions.

**Example: Program to find the factorial of a given number using recursion.**

```
#include<stdio.h>
int fact (int n)
{
 if(n ==0)
 return 1;
 return n * fact(n-1) ;
}
int main()
{
 int N, result;
 printf("Enter the value of N\n");
 scanf("%d", &N);
 result = fact(N);
 printf("Factorial of %d = %d\n" , N, result);
 return 0;
}
```

**VII. Multi-Dimensional Array**

If an array contains more than one subscript then it is called as multi-dimensional array.

**Two-Dimensional Arrays:**

If an array contains two subscripts then it is called as two-dimensional array. The elements in two-dimensional array are arranged in matrix or table form.

**Declaration and definition of Two-Dimensional array**

An array must be declared and defined before it can be used. Array declaration and definition tells the compiler

- Name of the array
- The size or the number of elements in the array

- The type of each element of the array
- Total amount of memory to be allocated to the array

**The size of the array is a constant and must have a value at compilation time.**

The declaration format or syntax of declaration is given below;

**Type                      array\_name [ROWSIZE][COLSIZE] ;**

Total number of elements in a two-dimensional array is calculated by multiplying ROWSIZE and COLSIZE.

i.e.,      No of elements                      = ROWSIZE \* COLSIZE

Total Memory Allocated                      = ROWSIZE \* COLSIZE \* sizeof(Type)

**Where,**

**Type** -                      refers to the type of each element of the arrays which can be int, float, char, double, short, long, etc.

**array\_name** -                      refers to the name of the array to be specified with the help of rules of identifier

**ROWSIZE** -                      refers to the size or number of rows of the array.

**COLSIZE** -                      refers to the size or number of columns of the array.

These **ROWSIZE** and **COLSIZE** can be a symbolic constants, integer constants or integer arithmetic expressions.

**Examples:**

- declare and define an array by the name **matrix** which contains **3 rows and 4 columns of integer type**.

```
int matrix[3][4];
```

- declare and define an array called **string** which contains 5 strings each of 10 characters in size.

```
char string[5][10];
```

## Accessing Elements in Two-dimensional Arrays

C uses an **index** to access individual elements in an array. The index must be an integral value or an expression that evaluates to an integral value.

For example, consider the following array

```
int matrix[3][3];
```

Pictorial representation of two-dimensional matrix is given below

| Column Index<br>Row Index | 0            | 1            | 2            |
|---------------------------|--------------|--------------|--------------|
| 0                         | matrix[0][0] | matrix[0][1] | matrix[0][2] |
| 1                         | matrix[1][0] | matrix[1][1] | matrix[1][2] |
| 2                         | matrix[2][0] | matrix[2][1] | matrix[2][2] |

To access **first element of first row** of the array **matrix**, we can use **matrix[0][0]**, since the index of first row is ZERO (0) and first column is ZERO (0).

Similarly, **second element of first row** of the array can be accessed by **matrix[0][1]**, **third element of second row** can be accessed by **matrix[1][2]**, etc.

## Storing Values in two-dimensional Arrays

Declaration and definition only reserve space for the elements in the array. No values are stored.

If we want to store values in the array, we can use the following different methods of initialization

- initialize the elements (**Initialization**)
- read values from the keyboard (**Inputting**)
- assign values to each individual element (**Assignment**)

### Initialization of two-dimensional array

Providing a value for each element of the array is called as initialization. The list of values must be enclosed in curly braces.

**NOTE: It is a compile time error if we provide more values than the size of the array.**

Example 1: Initializing all elements. All elements are initialized sequentially.

```
int numbers[3][2] = { 3, 23, 17, 8, 19, 7};
```

In this example, two-dimensional arrays **numbers** is initialized as given below

| Column Index \ Row Index | 0  | 1  |
|--------------------------|----|----|
| 0                        | 3  | 23 |
| 1                        | 17 | 8  |
| 2                        | 19 | 7  |

Example 2: Initializing all elements but each row is initialized independently.

```
int Array[3][3] = { { 2, 3, 4},
 {12, 8, 6},
 {6, 3, 16}
 };
```

In this example, all elements are initialized row by row as given below.

| Column Index \ Row Index | 0  | 1 | 2  |
|--------------------------|----|---|----|
| 0                        | 2  | 3 | 4  |
| 1                        | 12 | 8 | 6  |
| 2                        | 6  | 3 | 16 |

Example 3: Partial initialization.

```
int a[3][3] = { { 1 }, { 5, 2 }, { 6 } };
```

In this example, only specified elements are initialized row by row and uninitialized elements are set to ZERO by default as given below.

| Column Index \ Row Index | 0 | 1 | 2 |
|--------------------------|---|---|---|
| 0                        | 1 | 0 | 0 |
| 1                        | 5 | 2 | 0 |
| 2                        | 6 | 0 | 0 |

### Inputting Values

Another way to fill the array is to read the values from the keyboard or file by using scanf statement or fscanf statement.

For Example,

```
int numbers[3][2];
int i, j ;
for(i=0;i<3;i++)
 for(j=0;j<2;j++)
 scanf("%d", &numbers[i][j]);
```

In this example, scanf function is repetitively called for  $3 \times 2 = 6$  times and reads the values for all elements of the array. Since the starting index of row and column is ZERO, so index **i** and **j** are initialized to 0 in for loop and the condition  $i < 3$  iterates 3 times i.e., the number of rows and the condition  $j < 2$  iterates for 2 times i.e., the number of columns.

### Assigning values

We can assign values to individual elements using the assignment operator. Any value that reduces to the proper type of array can be assigned to an individual array element.

For Example,

```
int numbers[3][3];
numbers[0][0] = 10; //assigns first element of first row to 10
numbers[1][2] = 4; //assigns third element of second row to 4
```

## VIII. Using Arrays with Functions

To process arrays in a large program, we have to be able to pass them to functions. We can pass arrays in two ways:

- a. Passing individual elements
- b. Passing the entire array

### a. Passing individual elements

As we know that there are two parameter passing mechanisms, we can pass individual elements by either data values or by passing their addresses. **The default method is pass-by-value.**

#### Passing Data Values

We pass data values of the individual elements of the array. The actual parameters in function call are individual elements of the array whose values are copied to the formal parameters.

For example:

```
int arr[10]; // creates an array of integer numbers
fun (arr[3]); // invokes a function fun with 4th element of array as parameter

void fun(int x) //the value of 4th element of array is copied to
{ // formal parameter x

 Process x;

}
```

#### Passing Addresses

We pass addresses of the individual elements of the array. The actual parameters in function call are individual elements of the array whose addresses are copied to the formal parameters.

For example:



```
int arr[10]; // creates an array of integer numbers
fun (&arr[3]); // invokes a function fun with address of 4th element of array as
 // parameter

void fun(int *x) //the address of 4th element of array is copied to
{ // formal parameter *x

 Process x;

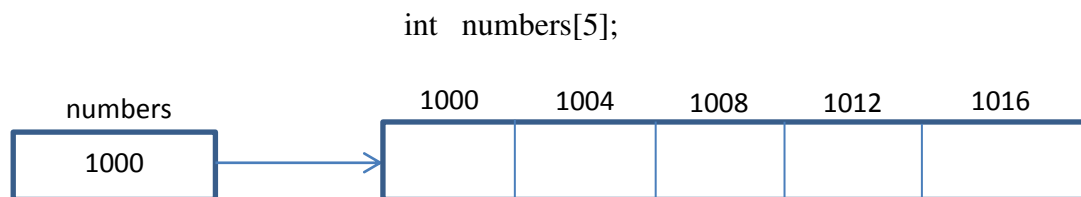
}
```

### b. Passing the entire array

Passing individual elements of the array is not a feasible method since it requires a lot of memory and time to process large array. For example, if an array consists of 20000 elements then if we pass individual elements to the function then another 20000 memory blocks are to be allocated in function to store the values of each element of the array. So instead of passing individual elements, starting address of whole array can be passed. Hence, for passing the entire array, **the default method of parameter passing mechanism is pass-by-address or reference.**

**NOTE: In C, the name of the array is a primary expression whose value is the address of the first element in the array.**

A pictorial representation of above NOTE is shown below.



From the picture, we can understand that the name of array numbers is having the starting address i.e., 1000 in it. With the help of this address we can go through each and every element of the array. **Hence, to pass the entire array to the function, we can pass only name of the array** which contains starting address.

To pass the whole array, we simply use the array name as actual parameter in a function call. In function declaration or definition, we can declare a formal parameter in two different ways.

First, it can use an array declaration with an empty set of brackets in the parameter list.

Secondly, it can specify that the array parameter is a pointer.

**Any modification in the function definition by using formal parameter will modify actual parameter since formal parameter is a reference to the actual parameter.**

For example:

```
void fun (int num[],...);
void fun (int *ptr, ...);
```

**Example Program: C program to generate N Fibonacci numbers**

Fibonacci numbers or series starts with either 0 and 1 or 1 and 1 then the next number in the series is sum of two previous terms.

Therefore,

Fibonacci number can be either 0, 1, 1, 2, 3, 5, 8, 13, 21, ... or it can be 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

We will write a program for first series i.e., 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
#include<stdio.h>
void fibonacci(int *f, int N);
int main()
{
 int i, N, fib[100];
 printf("Enter the number of elements to be generated\n");
 scanf("%d", &N);
 fibonacci(fib, N); // in this function call, starting address of fib array is copied to
 // formal parameter *f which is pointer in declaration
 printf("The Fibonacci Numbers are\n");
 for(i=0;i<N;i++)
 printf("%d\t", fib[i]); // prints the fibonacci numbers generated in
 // function definition

 return 0;
}
void fibonacci(int *f, int N)
{
 int i;
 f[0] = 0;
 f[1] = 1;
 for(i=2;i<N;i++)
 f[i] = f[i-1] + f[i-2] ;
}
```

**IX. Declaring, Initializing, Printing, and Reading Strings****Definition of String:**

A C string is an array of characters terminated by null (\0) character. The \0 (null) character indicates the end of string. String in C is always variable in length.

Following example illustrates how a string is stored in memory.



What is important here is that the string "Hello" is stored in an array of characters that ends with null (\0) character.

**NOTE 1: Empty string is represented by "" which requires one byte of memory.**

**NOTE 2: null (\0) character is known as delimiter.**

**Why null (\0) character?**

String is not a data type but it is a data structure and its implementation is logical not physical. Since string length is variable in C, to identify the end of string this null character is used.

**Definition of String Literal or constant**

A string literal is also known as string constant, is a sequence of characters enclosed in a pair of double quotes.

For example:

“Welcome to C programming”

“Hello”

“abcd”

**When string literals are used in a program, C automatically creates an array of characters, initializes it to a null-terminated string, and stores it and remembering its address.**

**Declaring a String:**

Like other variables, the string has to be declared before it is used in the program. Since string is a sequence of characters, it is to be declared as array of characters.

For example:

If we want a string declaration for an eight-character string, including its delimiter then it must be declared as shown below

```
char string[9];
```

Using this declaration, the compiler allocates 9 bytes of memory for the variable string.

**Initializing a String:**

We can initialize a string the same way that we initialize any other variable when it is defined.

Initialization of string can be done in different ways

**1. Initialization using string constant**

```
char str[9] = “Good Day”; // in this case, the value is a string constant
```

**2. Initialization without size using string constant**

We do not need to specify the size of the array if we initialize it when it is defined.

```
char str[] = “January”;
```

**3. Initialization using array of characters**

```
char str[9] = { ‘G’, ‘o’, ‘o’, ‘d’, ‘ ’, ‘D’, ‘a’, ‘y’, ‘\0’ };
```

**NOTE 1: in this example, we must ensure that the null character is at the end of the string.**

**NOTE 2: Strings and Assignment Operator: One string cannot be assigned to another string using assignment operator. It is a compiler time error if we try to assign.**

For example:

```
char str1[20] = “Hello”;
```

```
char str2[20];
```

```
str2 = str1 ; // it's a compiler time error: one string cannot be
copied to another string using assignment operator.
```

### Printing and Reading of Strings:

There are several functions to read the string from the keyboard or file and to display or print the string onto the monitor, printer or a file. They are discussed in the section **String Input and Output Functions**.

## X. String Input and Output Functions

C programming treats all the devices as files. So devices such as the display or monitor are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

| Standard File   | File Pointer | Device   |
|-----------------|--------------|----------|
| Standard input  | stdin        | keyboard |
| Standard output | stdout       | screen   |
| Standard error  | stderr       | screen   |

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the keyboard and how to print the result on the screen.

C provides two basic ways to read and write strings.

1. **Formatted Input and Output Functions:** the functions which helps to read and write the data in the required format are known as formatted input and output functions
  - a. Formatted Input Functions: `scanf()` and `fscanf()`
  - b. Formatted Output Functions: `printf()` and `fprintf()`

### Formatted Input Functions: `scanf()`

We read strings from the keyboard using the read-formatted function i.e. `scanf()`.

The **conversion code** for a string is 's'. Therefore, **format string or control string** to read a string is "%s".

`Scanf()` do the following work when we enter string from the keyboard

- A. First, it removes the whitespaces which appear before the string
- B. Once it finds a character, then it starts reading characters until it finds a whitespace and stores in array in order.
- C. Then it ends the string with a null character.

For example:

```
char str[20];
scanf("%s", str); // & operator must not be used since str contains starting
 // address of string
```

If we enter the following string as input from keyboard, then it reads HELLO as string, terminates it with null character and stores in **str**. Therefore, `str[20]` will be "HELLO\0".

|       |       |       |   |   |   |   |   |       |   |   |   |   |   |                |
|-------|-------|-------|---|---|---|---|---|-------|---|---|---|---|---|----------------|
| Space | Space | Space | H | E | L | L | O | Space | W | O | R | L | D | Press<br>Enter |
|-------|-------|-------|---|---|---|---|---|-------|---|---|---|---|---|----------------|

In this example, the `scanf()` has skipped 3 leading white spaces and started reading from the character 'H' until a white space. Remaining characters will not be read and left in buffer.

**NOTE: The string conversion code skips whitespace.**

#### **Drawback of scanf():**

The main drawback of this `scanf()` to read a string is that as soon as it encounters a white space it stops reading the string. Hence, it is not possible to read a string with multiple words separated by white space (sentence for example).

### **Formatted Output Functions: printf()**

We print strings on monitor using the print-formatted function i.e. `printf()`.

The **conversion code** for a string is 's'. Therefore, **format string or control string** to print a string is "%s".

For example:

```
char str[20] = "HELLO";
printf("%s", str); // prints the content of string str on screen i.e., HELLO
```

#### **Example program to read and display a string using scanf and printf functions**

```
#include<stdio.h>

int main()
{
 char str[20];
 printf("Enter a String\n");
 scanf("%s", str);
 printf("String entered by You is : %s \n", str);
 return 0;
}
```

The program reads the string entered from the keyboard using `scanf` function and the same string will be displayed on screen using `printf` function.

2. **Unformatted Input and Output Functions:** the functions which reads and writes the data without any format are known as unformatted input and output functions
  - a. Unformatted Input Functions: `gets()`, `getchar()`, `getc()`, and `fgetc()`
  - b. Unformatted Output Functions: `puts()`, `putchar()`, `putc()`, and `fputc()`

### **Unformatted Input and output Functions: getchar() and putchar()**

The **int** `getchar(void)` function reads the next available character from the keyboard and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int** `putchar(int c)` function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen.

```
#include <stdio.h>
int main()
{
 int c;
 printf("Enter a value :");
 c = getchar();

 printf("\nYou entered: ");
 putchar(c);

 return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows

./a.out

**Enter a value : I am reading a character using getchar**

**You entered : I**

From the output, we can understand that the `getchar` function reads only single character at a time i.e I and stores it in variable `c`. So only, `putchar` function prints value of variable `c` i.e. I on the screen.

## Unformatted input and output functions: `getc()` and `putc()`

### `getc()`

The C library function **`int getc(FILE *stream)`** gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

#### Declaration

Following is the declaration for `getc()` function.

```
int getc(FILE *stream)
```

#### Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream on which the operation is to be performed. It should be **`stdin`** if we want to read data from keyboard.

#### Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

### `putc()`

The C library function **`int putc(int ch, FILE *stream)`** writes a character (an unsigned char) specified by the argument **`ch`** to the specified stream and advances the position indicator for the stream.

### Declaration

Following is the declaration for `putc()` function.

```
int putc(int ch, FILE *stream)
```

### Parameters

- **char** -- This is the character to be written. The character is passed as its int promotion.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the character is to be written. It should be **stdout** to write on screen.

### Return Value

This function returns the character written as an unsigned char cast to an int or EOF on error.

### Example

The following example shows the usage of `getc()` and `putc()` function.

```
#include<stdio.h>
int main()
{
 char c;
 printf("Enter character: ");
 c = getc(stdin);
 printf("Character entered: ");
 putc(c, stdout);
 return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Enter character: A
Character entered: A
```

## Unformatted Input Functions: `gets()` and `puts()`

The **char \*gets(char \*s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File). It makes null-terminated string after reading. Hence it is also known as **line-to-string** input function.

The **int puts(const char \*s)** function writes the null-terminated string 's' from memory and a trailing newline to **stdout**. This function is also known as **string-to-line** output function.

```
#include <stdio.h>
int main()
{
 char str[100];
 printf("Enter a value :");
 gets(str);

 printf("\nYou entered: ");
 puts(str);
}
```

```
return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows

./a.out

**Enter a value : I am reading a line**

**You entered : I am reading a line**

From the output, we can understand that the gets function reads a line at a time i.e **I am reading a line** and stores it in variable str. So only, puts function prints value of variable str i.e. **I am reading a line** on the screen.

### **Program: C program to read a sentence and count the frequency of VOWELS and total count of consonants.**

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

```
int main()
```

```
{
```

```
 char sentence[100];
```

```
 int vowelA=0, vowelE=0, vowelI=0, vowelO=0, vowelU=0, cons=0;
```

```
 printf("Enter a sentence\n");
```

```
 gets(sentence);
```

```
 i=0;
```

```
 while((ch=sentence[i++])!= '\0')
```

```
 {
```

```
 switch(ch)
```

```
 {
```

```
 case 'a':
```

```
 case 'A': vowelA++;
```

```
 break;
```

```
 case 'e':
```

```
 case 'E': vowelE++;
```

```
 break;
```

```
 case 'i':
```

```
 case 'I': vowelI++;
```

```
 break;
```

```
 case 'o':
```

```
 case 'O': vowelO++;
```

```
 break;
```

```
 case 'u':
```

```
 case 'U': vowelU++;
```

```
 break;
```

```
 default: cons++;
```

```
 }
```

```
 }
```



```
printf("Frequency of Vowel A = %d\n", vowelA);
printf("Frequency of Vowel E = %d\n", vowelE);
printf("Frequency of Vowel I = %d\n", vowelI);
printf("Frequency of Vowel O = %d\n", vowelO);
printf("Frequency of Vowel U = %d\n", vowelU);
printf("Total count of Consonants = %d\n", cons);
return 0;
}
```

## XI. String Manipulation Functions or String Handling Functions

Manipulation includes performing various operations on strings according to the need of a problem. Hence, in C programming language, there are a lot of functions that can be used to manipulate or to handle strings. The strings manipulations involve combining or concatenating strings, copying the content of one string to another string, comparing two strings, calculating the length of a string, finding a character or a substring in another string, and converting a string to long or double value that can be used for computations.

### List of Built-In Functions available in C

- a. `strlen()` - calculates the length of string
- b. `strcpy()` - copies one string to another string
- c. `strncpy()` - copies first N characters of one string to another string
- d. `strcat()` - concatenates or joins two strings
- e. `strncat()` - concatenates or joins first N characters of one string with another string
- f. `strcmp()` - compares two strings
- g. `strncmp()` - compares first N characters of two strings
- h. `strchr()` - find first occurrence of a given character in string
- i. `strrchr()` - find last occurrence of a given character in string
- j. `strstr()` - find first occurrence of one string in another string
- k. `strtok()` - parse or split the string into tokens using specified delimiters

**NOTE: to use any of these functions in C program, one must include header file `string.h` since it contains all of these string handling functions.**

- a. `strlen()` - calculates the length of string

### Definition:

The `strlen` function calculates the length, in bytes, of a given string. This calculation does not include the null character. The function returns length of string.

### Syntax:

Variable = `strlen(string)`;

Where string can be C string or it can be String literal or constant.

### Example 1:

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

int length;
length = strlen(str);
```

**Output:**

**length=5** since, **strlen()** returns length of string i.e. 5 since the string constant "Hello" is having 5 characters.

**Example 2:**           int len;

                    len = strlen("Hello World");

**Output:**

**len=11**, since there are 11 characters in string constant "Hello World"

**b. strcpy()**       - copies one string to another string

**Definition:**

The strcpy function copies characters from source string to destination string up to and including the terminating null character. The strcpy function returns destination string.

**Syntax:**

**strcpy(destination, source);**

where,

destination   - must be array of character type

source       - can be either C string or string literal

**Example 1:**

```
char src[20] = "Hello";
char dest[20];
strcpy(dest, src); // copies content of src to dest
printf("dest = %s\n", dest);
```

**Output:**

**dest = Hello** since strcpy copies content of src i.e "Hello" to dest. Therefore, dest will be "Hello".

**Example 2:**

```
char str[20];
strcpy(str, "Hello World"); // copies string literal to str
printf("str = %s\n", str);
```

**Output:**

**str = Hello World** since strcpy copies string constant i.e "Hello World" to str. Therefore, str will be "Hello World".

**c. strncpy()**       - copies first N characters of one string to another string

**Definition:**

The strncpy function copies N characters from source string to destination string up to and including the terminating null character if length of source string is less than N. The function returns destination string.

**Syntax:**

**strncpy(destination, source, N);**

where,

destination   - must be array of character type

source       - can be either C string or string literal

N - must be an integer value

**Example 1:**

```
char src[20] = "Hello";
char dest[20];
strcpy(dest, src, 2); // copies first 2 characters of src to dest
printf("dest = %s\n", dest);
```

**Output:**

**dest = He** since strcpy copies first 2 characters of src to dest. Therefore, dest will be "He".

**Example 2:**

```
char str[20];
strcpy(str, "Hello World", 7); // copies first 7 characters of string literal to str
printf("str = %s\n", str);
```

**Output:**

**str = Hello W** since strcpy copies first 7 characters of string constant to str. Therefore, str will be "Hello W".

**Example 3:**

```
char src[20] = "Hello";
char dest[20];
strcpy(dest, src, 20); // copies first 20 characters of src to dest
printf("dest = %s\n", dest);
```

**Output:**

**dest = Hello** since the length of string src is 5 which is less than 20 so it copies entire string in src to dest. Therefore, dest will be "Hello".

**d. strcat()** - concatenates or joins two strings

**Definition:**

The strcat function concatenates or appends or joins source string with destination string. All characters from source string are copied including the terminating null character.

**Syntax:**

**strcat(destination, source);**

**where,**

destination - must be an array

source - can be either C string or string literal

**Example:**

```
char string1[20] = "Hello";
char string2[20] = "World";
strcat(string1, string2);
printf("String 1 = %s\n", string1);
```

**Output:**

**String 1 = HelloWorld** since strcat concatenates content of string2 with string1. Therefore, string1 = "HelloWorld".

- e. **strncat()** - concatenates or joins first N characters of one string with another string

**Definition:**

The strncat function concatenates or appends or joins first N characters from source string to destination string. All characters from source string are copied including the terminating null character if length of source string is less than or equal to N.

**Syntax:**

**strcat(destination, source, N);**

**where,**

- |             |                                            |
|-------------|--------------------------------------------|
| destination | - must be an array                         |
| source      | - can be either C string or string literal |
| N           | - must be an integer value                 |

**Example 1:**

```
char string1[20] = "Hello";
char string2[20] = "World";
strcat(string1, string2, 3);
printf("String 1 = %s\n", string1);
```

**Output:**

**String 1 = HelloWor** since strcat concatenates first 3 characters of string2 with string1. Therefore, string1 = "HelloWor".

**Example 2:**

```
char string1[20] = "Hello";
char string2[20] = "World";
strcat(string1, string2, 20);
printf("String 1 = %s\n", string1);
```

**Output:**

**String 1 = HelloWorld** since length of string2 is less than 20 so entire source string is concatenated with string1. Therefore, string1 = "HelloWorld".

- f. **strcmp()** - compares two strings

**Definition:**

The strcmp function compares the contents of string1 and string2 and returns an integer value indicating their relationship.

**Syntax:**

**strcmp(string1, string2)**

**where,**

string1 and string2 can be either C string or String constant

**Return value:**

- If string1 and string2 are equal then it returns ZERO
- If string1 is less than string2 then it returns negative value(<0)
- If string1 is greater than string2 then it returns positive value(>0)

**Example 1:**

```
char string1[20] = "Hello";
char string2[20] = "Hello";
n = strcmp(string1, string2);
```

**Output:**

**n = 0** since string1 and string2 are equal

**Example 2:**

```
char string1[20] = "Hello";
char string2[20] = "World";
n = strcmp(string1, string2);
```

**Output:**

**n = less than 0** since string1 is less than string2

**Example 3:**

```
char string1[20] = "Welcome";
char string2[20] = "Hi";
n = strcmp(string1, string2);
```

**Output:**

**n = greater than 0** since string1 is greater than string2

**g. strncmp() - compares first N characters of two strings****Definition:**

The strncmp function compares first N characters of string1 and string2 and returns an integer value indicating their relationship.

**Syntax:**

**strncmp(string1, string2, N)**

**where,**

string1 and string2 can be either C string or String constant and N must be an integer value

**Return value:**

- If first N characters of string1 and string2 are equal then it returns ZERO
- If first N characters of string1 is less than string2 then it returns negative value(<0)
- If first N characters of string1 is greater than string2 then it returns positive value(>0)

**Example 1:**

```
char string1[20] = "Hello";
char string2[20] = "Hello";
X = strncmp(string1, string2, 3);
```

**Output:**

**X = 0** since first 3 characters of string1 and string2 are equal

**Example 2:**

```
char string1[20] = "Hello";
char string2[20] = "Horld";
X = strcmp(string1, string2, 2);
```

**Output:**

**X = less than 0**      since first 2 characters of string1 is less than string2

**Example 3:**

```
char string1[20] = "Welcome";
char string2[20] = "Hi";
X = strcmp(string1, string2, 1);
```

**Output:**

**n = greater than 0**      since first 1 character of string1 is greater than string2

**h. strchr()      - find first occurrence of a given character in string****Definition:**

The strchr function searches string for the first occurrence of a specified character. The null character is also included in the search. The function returns a pointer to the first occurrence of given character in string or null pointer if no matching character is found.

**Syntax:**

**strchr(string, ch)**

**where,**

|        |                                      |
|--------|--------------------------------------|
| string | - can be C string or string constant |
| ch     | - character to be searched           |

**Example:**

```
char *s;
char buf[] = "This is a test";
s = strchr (buf, 't');
if (s != NULL)
 printf ("found a 't' at %s\n", s);
```

**Output:**

It will produce following result

**found a 't' at test**

**i. strrchr()      - find last occurrence of a given character in string****Definition:**

The strrchr function searches string for the last occurrence of a specified character. The null character is also included in the search. The function returns a pointer to the last occurrence of given character in string or null pointer if no matching character is found.

**Syntax:**

**strrchr(string, ch)**

**where,**

|        |                                      |
|--------|--------------------------------------|
| string | - can be C string or string constant |
| ch     | - character to be searched           |

**Example:**

```
char *s;
char buf [] = "This is testing";
s = strchr (buf, 't');
if (s != NULL)
 printf ("found a 't' at %s\n", s);
```

**Output:**

It will produce following result

**found a 't' at ting**

**j. strstr() - find first occurrence of one string in another string****Defintion:**

The strstr function locates the first occurrence of the string2 in the string1 and returns a pointer to the beginning of the first occurrence in string1 if matching string2 is found otherwise null pointer.

**Syntax:**

**strstr( string1, string2);**

**where,**

string1 and string2 can be C string or string constant

**Example:**

```
char s1[] = "My House is small";
char *s2;
s2 = strstr(s1, "House");
printf ("Returned String : %s\n", s2);
```

**Output:**

It will produce the following result

**Returned String : House is small**

**k. strtok() - parse or split the string into tokens using specified delimiters****Definition:**

The strtok function is used to locate substrings known as tokens in a string. Its most common use is to split or parse the given string into tokens using one or more delimiters.

**Syntax:**

**strtok(string, delimiters)**

**where,**

string - can be either C string or string literal which is to be parsed

delimiters - one or more characters to be used to split or parse the given string

**Example:**

```
char| string[] = "ONE,TWO-THREE";
strtok(string, ",-");
```

In this example, the two delimiters specified are comma(,) and hyphen(-) hence the given string "ONE,TWO-THREE" is parsed or split into three substrings namely "ONE" , "TWO" and "THREE".

**Output:**

**"ONE"**

“TWO”

“THREE”

## XII. Array of Strings

As we know the definition of string is that it is an array of characters, then array of strings can be defined as array of array of characters. Hence, array of strings is basically an application of two-dimensional arrays. Each string is independent and at the same time they are grouped together through the array.

For example, if we need to store seven days of the week in the array then we have to arrange these as array of strings since we have to store seven days of the week and each day of the week is again an array of characters so it is two-dimensional array.

### Declaration of array of strings

```
#define days 7
#define size 15
char week[days][size];
```

In this example, two-dimensional array called week is been created with seven days and each day can have maximum of 15 characters.

### Initialization of array of strings

#### a. Initialization with SIZE

```
#define days 7
#define size 15
char week[days][size] = {"Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday"};
```

Pictorial representation of array of string is shown below

|   |   |   |   |   |   |   |    |    |    |    |
|---|---|---|---|---|---|---|----|----|----|----|
| → | M | O | N | D | A | Y | \0 |    |    |    |
| → | T | U | E | S | D | A | Y  | \0 |    |    |
| → | W | E | D | N | E | S | D  | A  | Y  | \0 |
| → | T | H | U | R | S | D | A  | Y  | \0 |    |
| → | F | R | I | D | A | Y | \0 |    |    |    |
| → | S | A | T | U | R | D | A  | Y  | \0 |    |
| → | S | U | N | D | A | Y | \0 |    |    |    |

#### b. Initialization without SIZE

```
#define days 7
char week[days][] = {"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"};
```

#### c. Inputting the values through keyboard

```
#define days 7
#define size 15
char week[days][size];
int i;
for(i=0;i<days;i++)
 scanf("%s",week[i]);
```



**Questions appeared in previous question papers****1. Write a C program to read N integers into an array A and find****i) the sum of odd numbers****ii) the sum of even numbers****iii) the average of all numbers****Output the results computed with appropriate headings.****Solution:**

```
#include<stdio.h>
int main()
{
 int i, a[100], osum=0, esum=0, N;
 float average;
 printf("\n Enter the value of N:");
 scanf("%d",&N);
 printf("\n Enter %d integers \n",N);
 for(i=0;i<N;i++)
 scanf("%d", &a[i]);
 for(i=0;i<N;i++)
 {
 if(a[i]%2==0)
 esum = esum +a[i];
 else
 osum = osum +a[i];
 }
 average = (float)(esum+osum)/N;
 printf("\n Sum of even numbers = %d\n",esum);
 printf("\n Sum of odd numbers = %d\n", osum);
 printf("\n The average of all numbers = %f\n",average);
 return 0;
}
```

**2. Write a C Program to concatenate two strings without using built in function strcat().**

```
#include<stdio.h>
#include <string.h>
void strconcat(char dest[],char src[])
{
 int len=strlen(dest),i=0;
 // start copying from the index len of dest from src
 while (src[i]!='\0')
 {
 dest[len]=src[i];
 len++;
 i++;
 }
}
```

```
 }
 dest[len]='\0';
}
int main(void)
{
 char dest[20]="ABCD",src[20]="EFG";
 strconcat(dest,src);
 printf("After concatenating the strings, destination string is %s\n",dest);
 return 0;
}
```

**3. Write a C Program to find length of a string without using built in function strlen().**

```
#include<stdio.h>
#include <string.h>
int strlenlength(char src[])
{
 int i=0;
 // start finding the length from first char to last char
 while (src[i]!='\0')
 i++;
 return i;
}
int main(void)
{
 char src[20];
 printf("Enter a string\n");
 scanf("%s", src);
 printf("Length of %s string=%d\n", strlenlength(src));
 return 0;
}
```

**4. Write a C program to find cube of a number using function.**

```
#include<stdio.h>
int cube(int n)
{
 return n*n*n;
}
int main(void)
{
 int n;
 printf("Enter the number whose cube has to be found\n");
 scanf("%d",&n);
 printf("The cube of the number %d is %d\n",n,cube(n));
 return 0;
}
```

```
}
```

**5. Write a C program using recursion to print prime numbers between 1 and 100.**

```
#include<stdio.h>
int isprime(int n, int p)
{
 if (p==1)
 return 1;
 else if (n%p==0)
 return 0;
 else
 return isprime(n,p-1);
}
int main(void)
{
 int i;
 printf("The prime numbers are :");
 for (i=2;i<100;i++)
 if (isprime(i, i/2)==1)
 printf("%d ",i);
 return 0;
}
```

**6. Write a C Program to find the greatest number from a given one dimensional array.**

```
#include<stdio.h>
int main()
{
 int arr[20], i, large, n;
 printf("Enter number of elements\n");
 scanf("%d", &n);
 printf("Enter %d elements\n", n);
 for(i=0; i<n; i++)
 scanf("%d",&arr[i]);
 printf("The largest element is: ");
 large = a[0];
 for(i=1; i<n; i++)
 {
 if(a[i] > large)
 large = a[i];
 }
 printf("%d", large);
 return 0;
}
```

7. Write a C function `isprime(num)` that accepts an integer argument and returns 1 if the argument is prime, a 0 otherwise. Write a C program that invokes this function to generate prime numbers between the given ranges.

OR

Write a C Program to list prime numbers in the given range of numbers.

**Solution:**

```
#include<stdio.h>
int isprime(int n, int p)
{
 if (p==1)
 return 1;
 else if (n%p==0)
 return 0;
 else
 return isprime(n,p-1);
}
int main(void)
{
 int i, num1, num2;
 printf("Enter range of numbers\n");
 scanf("%d%d", &num1, &num2);
 printf("The prime numbers are :");
 for (i=num1;i<=num2;i++)
 if (isprime(i, i/2)==1)
 printf("%d ", i);
 return 0;
}
```

8. Write a C program to evaluate polynomial  $f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0$ .

**Solution:**

```
#include<stdio.h>
int main()
{
 int n, i, sum=0, a[10], x;
 printf("\n Enter the value of n:");
 scanf("%d",&n);
 printf("\n Enter the %d co-efficient:\n",n+1);
 for(i=0;i<=n;i++)
 scanf("%d",&a[i]);
 printf("\n Enter value of x :");
 scanf("%d",&x);
 for(i=n; i>0; i--)
 sum=(sum+a[i])*x;
```

```
sum = sum + a[0];
printf("\n value of sum is :%d \n",sum);
return 0;
}
```

**9. Write a C program to read a matrix of size MxN and find the following**

- I. The sum of elements of each row**
- II. The sum of elements of each column**
- III. Total sum of all elements of matrix**

```
#include<stdio.h>
int main()
{
 int a[10][10], i, j , rsum, csum, tsum=0, m, n;
 printf("\n Enter the order of matrix:");
 scanf("%d%d",&m, &n);
 printf("\n Enter elements of matrix \n");
 for(i=0; i<m; i++)
 for(j=0; j<n; j++)
 scanf("%d", &a[i][j]);
 for(i=0; i<m; i++)
 {
 rsum=0;
 csum=0;
 for(j=0; j<n; j++)
 {
 rsum = rsum +a[i][j];
 csum = csum + a[j][i];
 }
 tsum = tsum + rsum
 printf("Sum of %d row = %d\n", i+1, rsum);
 printf("Sum of %d columns = %d\n", i+1, csum);
 }
 printf("Total sum of all elements = %d\n", tsum);
 return 0;
}
```

**10. Write a C program to find the largest element in a two dimensional array.**

```
#include<stdio.h>
int main()
{
 int i, j, a[10][10], large, m, n;
 printf("\n Enter the order of matrix:");
 scanf("%d%d",&m, &n);
 printf("\n Enter elements of matrix \n");
```

```
for(i=0; i<m; i++)
 for(j=0; j<n; j++)
 scanf("%d", &a[i][j]);
large = a[0][0];
for(i=0; i<m; i++)
{
 for(j=0; j<n; j++)
 {
 if(a[i][j]>=large)
 large = a[i][j];
 }
}
printf("Largest element in a matrix = %d\n", large);
return 0;
}
```

**11. Write a C program to find the addition of two matrices a and b. Print the resultant matrix.**

```
#include<stdio.h>
int main()
{
 int i, j, a[10][10], b[10][10], c[10][10], m, n;
 printf("\n Enter the order of matrix a:");
 scanf("%d%d",&m, &n);
 printf("\n Enter elements of matrix A\n");
 for(i=0; i<m; i++)
 for(j=0; j<n; j++)
 scanf("%d", &a[i][j]);
 printf("\n Enter elements of matrix B\n");
 for(i=0; i<m; i++)
 for(j=0; j<n; j++)
 scanf("%d", &b[i][j]);
 for(i=0; i<m; i++)
 for(j=0; j<n; j++)
 c[i][j] = a[i][j] + b[i][j];
 printf("Resultant matrix C is:");
 for(i=0; i<m; i++)
 {
 for(j=0; j<n; j++)
 printf("%5d", c[i][j]);
 printf("\n");
 }
 return 0;
}
```

**12. Write a C program that reads N integer numbers and arrange them in ascending order using Bubble Sort technique.**

```
#include<stdio.h>
int main()
{
 int n, i, j, a[10], temp;
 printf("Enter the no of elements:\n");
 scanf("%d", &n);
 printf("\n Enter %d elements\n", n);
 for(i=0; i<n; i++)
 scanf("%d",&a[i]);
 printf("The original elements are:\n");
 for(i=0; i<n; i++)
 printf("%d\t", a[i]);
 for(i=0; i<n-1; i++)
 {
 for(j=0; j<n-1-i; j++)
 {
 if(a[i] > a[j])
 {
 temp = a[i];
 a[i] = a[j];
 a[j] = temp;
 }
 }
 }
 printf("\n The Sorted array is :\n");
 for(i=0; i<n; i++)
 printf("%d\n", a[i]);
 return 0;
}
```

**13. Write a C program to search a Name in a list of names using Binary searching Technique.**

```
#include<stdio.h>
int main()
{
 char arr[25][20], key[25];
 int i, n, low, high, mid, cond;
 printf("Enter the number of strings you want to enter\n");
 scanf("%d", &n);
 printf("Enter %d strings in alphabetical order\n", n);
 for (i=0; i<n; i++)
 scanf("%s", arr[i]);
 printf("Enter string to be searched\n");
 scanf("%s", key);
 low = 0;
 high = n-1;
 while(low <= high)
```

```
{
 mid = (low + high) / 2;
 if((cond = strcmp(key, arr[mid])) == 0)
 {
 printf("Key found at %d position\n", mid+1);
 return 0;
 }
 else if(cond < 0)
 high = mid - 1;
 else
 low = mid + 1;
}
printf("String %s is not found\n", key);
return 0;
}
```

**14. Write a C program to search a number in a list of numbers using Binary searching Technique.**

```
#include<stdio.h>
int main()
{
 int arr[25], key;
 int i, n, low, high, mid, cond;
 printf("Enter the number of elements you want to enter\n");
 scanf("%d", &n);
 printf("Enter %d numbers\n", n);
 for (i=0; i<n; i++)
 scanf("%d", &arr[i]);
 printf("Enter the number to be searched\n");
 scanf("%d", &key);
 //first sort all the elements in ascending order using bubble sort
 for(i=0; i<n-1; i++)
 {
 for(j=0; j<n-i-1; j++)
 {
 if(arr[j]>arr[j+1])
 {
 int temp = arr[j];
 arr[j] = arr[j+1];
 arr[j+1] = temp;
 }
 }
 }
 // search for a number using binary search
 low = 0;
 high = n-1;
 while(low <= high)
 {
 mid = (low + high) / 2;
```



```
 if(key == arr[mid])
 {
 printf("Key found at %d position\n", mid+1);
 return 0;
 }
 else if(key < arr[mid])
 high = mid - 1;
 else
 low = mid + 1;
 }
 printf("key %d is not found\n", key);
 return 0;
}
```

## MODULE 4: Structures and File Handling

### 1. What is structure data type? Explain.

#### Definition of Structures:

A structure is a collection of one or more variables of similar or dissimilar data types, grouped together under a single name i.e. A structure is a derived data type containing multiple variable of homogeneous or heterogeneous data types. The structure is defined using the keyword struct followed by an identifier. This identifier is called struct\_tag.

**Definition of Derived Data type:** it's a data type created by the using primitive data types or fundamental data types.

Examples: arrays, structures, pointers, union, etc.

The syntax and example is as follows:

**Syntax for Declaring and Defining Structure:** the declaration of structure defines a template for structure and its members. No memory allocation is done for the declared structure.

```
struct struct_tag
{
 type var_1;
 type var_2;
 .
 .
 .
 type var_n;
};
```

Here,

- The word struct is a keyword in C
- The item identified as struct\_tag is called a tag and identifies the structure later in the program
- Each type1, type2, ..., typen can be any valid C data type including structure type
- The item var\_1,var\_2,...,var\_n are the members or fields of the structure

Example:

```
struct student
{
 char name[20];
 int roll_number;
 float average_marks;
};
```

**Declaring Structure Variables:** declaring structure variables is nothing but creating instances of structures. Hence, actual memory allocation is done after declaring variables of structure type. Until and unless we create structure variables, it is not possible to store, process and access the members of the structure.

Total memory allocated for one variable of defined structure is sum of size of each member of the structure.

The syntax of declaring structure variables is shown below:

```
struct struct_tag v1,v2,v3,...,vn;
```

**Example:** struct student s1, s2, s3;

In the above statement, three variables s1, s2 and s3 are created of type struct student. Hence, separate memory allocation is done for each variable.

Therefore, total memory allocated for s1 = 20bytes (size of member **name**) + 4bytes (size of member **roll\_number**) + 8bytes(size of **average\_marks**) => 32bytes.

Similarly, for s2 another 32bytes of memory is allocated and for s3 also another 32bytes of memory is allocated.

The complete structure definition along with structure declaration is as shown below:

```
struct student
{
 char name[10];
 int roll_number;
 float average_marks;
};
struct student cse, ise; //creates two variables of
structure type
```

**Structure Initialization:** Assigning the values to the structure member fields is known as structure initialization. The syntax of initializing structure variables is similar to that of arrays i.e all the elements are enclosed within braces i.e { and } and are separated by commas. The appropriate values for each member of the structure must be provided in sequence. Partial initialization is also allowed but programmer must not omit initializers for middle members of the structure.

The syntax is as shown below:

```
struct struct_tag variable = { v1,v2,v3,...,vn};
```

Example : struct student cse = { "Raju",18, 87.5};

**Accessing Members of Structures ( Using The Dot (.) Operator ):**The member of structure is identified and accessed using the dot operator (.). The dot operator connects the member name to the name of its containing structure.

The general syntax is:

```
structure_variablename.membername;
```

Example: E1.name, here E1 is structure variable of structure employee and name is one of the member of structure employee.

**Structure assignment (using assignment operator):** the variable of one structure can be assigned to another variable of same structure. Every member value of one variable is copied to corresponding member of another variable.

**Example:**

```
struct student cse = { "Raju", 18, 87.5 }; // creates a variable cse
and initialized with values as specified in curly braces.
```

```
struct student ise; //creates another variable ise of structure student
type.
```

```
ise = cse;
```

in this statement, the value of name, roll\_number and average\_marks of cse are copied to name, roll\_number and average\_marks of ise respectively.

## 2. How structure is different from an array? Explain declaration of a structure with an example.

Both Structure and arrays are derived data types.

The major differences between structure and array are illustrated below:

| Arrays                                                                                      | Structures                                                                                                                   |
|---------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| It is a collection of homogeneous elements i.e All elements of same data type               | 1. It is a collection of homogeneous or heterogeneous elements i.e Members of a structure can belong to different data types |
| For arrays, the default parameter passing mechanism is call-by-address/reference technique. | For structure, the default parameter passing mechanism is call-by-value technique.                                           |
| Example : <code>int a[10]; char x[10];</code>                                               | Example:<br><pre>struct complex {     float real;     int    imaginary; };</pre>                                             |

**Syntax for Declaring and Defining Structure:** the declaration of structure defines a template for structure and its members. No memory allocation is done for the declared structure.

```
struct struct_tag
{
 type var_1;
 type var_2;
 .
 .
 .
 type var_n;
};
```

Here,

- The word `struct` is a keyword in C
- The item identified as `struct_tag` is called a tag and identifies the structure later in the program
- Each `type1`, `type2`, ..., `typen` can be any valid C data type including structure type

- The item var\_1,var\_2,...,var\_n are the members or fields of the structure

Example:

```
struct student
{
 char name[20];
 int roll_number;
 float average_marks;
};
```

**Declaring Structure Variables:** declaring structure variables is nothing but creating instances of structures. Hence, actual memory allocation is done after declaring variables of structure type. Until and unless we create structure variables, it is not possible to store, process and access the members of the structure.

Total memory allocated for one variable of defined structure is sum of size of each member of the structure.

The syntax of declaring structure variables is shown below:

```
struct struct_tag v1,v2,v3,...,vn;
```

**Example:** struct student s1, s2, s3;

In the above statement, three variables s1, s2 and s3 are created of type struct student. Hence, separate memory allocation is done for each variable.

Therefore, total memory allocated for s1 = 20bytes (size of member **name**) + 4bytes (size of member **roll\_number**) + 8bytes(size of **average\_marks**) => 32bytes.

Similarly, for s2 another 32bytes of memory is allocated and for s3 also another 32bytes of memory is allocated.

The complete structure definition along with structure declaration is as shown below:

```
struct student
{
 char name[10];
 int roll_number;
 float average_marks;
};
struct student cse, ise;
```

### 3. What is a structure? Explain the C syntax of structure declaration and initialization with an example.

#### Definition of Structures:

A structure is a collection of one or more variables of similar or dissimilar data types, grouped together under a single name i.e. A structure is a derived data type containing multiple variable of homogeneous or heterogeneous data types. The structure is defined using the keyword struct followed by an identifier. This identifier is called struct\_tag.

The syntax and example is as follows:

**Syntax for Declaring and Defining Structure:** the declaration of structure defines a template for structure and its members. No memory allocation is done for the declared structure.

```
struct struct_tag
{
 type var_1;
 type var_2;
 .
 .
 .
 type var_n;
};
```

Here,

- The word `struct` is a keyword in C
- The item identified as `struct_tag` is called a tag and identifies the structure later in the program
- Each `type1`, `type2`, ..., `typen` can be any valid C data type including structure type
- The item `var_1`, `var_2`, ..., `var_n` are the members or fields of the structure

Example:

```
struct student
{
 char name[20];
 int roll_number;
 float average_marks;
};
```

**Declaring Structure Variables:** declaring structure variables is nothing but creating instances of structures. Hence, actual memory allocation is done after declaring variables of structure type. Until and unless we create structure variables, it is not possible to store, process and access the members of the structure.

Total memory allocated for one variable of defined structure is sum of size of each member of the structure.

The syntax of declaring structure variables is shown below:

```
struct struct_tag v1,v2,v3,...,vn;
```

**Example:** `struct student s1, s2, s3;`

In the above statement, three variables `s1`, `s2` and `s3` are created of type `struct student`. Hence, separate memory allocation is done for each variable.

Therefore, total memory allocated for `s1` = 20bytes (size of member **name**) + 4bytes (size of member **roll\_number**) + 8bytes(size of **average\_marks**) => 32bytes.

Similarly, for `s2` another 32bytes of memory is allocated and for `s3` also another 32bytes of memory is allocated.

**Structure Initialization:** Assigning the values to the structure member fields is known as structure initialization. The syntax of initializing structure variables is

similar to that of arrays i.e all the elements are enclosed within braces i.e { and } and are separated by commas. The appropriate values for each member of the structure must be provided in sequence. Partial initialization is also allowed but programmer must not omit initializers for middle members of the structure.

The syntax is as shown below:

```
struct struct_tag variable = { v1,v2,v3,...,vn};
```

Example : 

```
struct student cse = { "Raju",18, 87.5};
```

#### 4. What are typedefinitions? What are the advantages of typedef? Explain with an example, how to create a structure using 'typedef'.

**Type definition:** The typedef is a keyword using which the programmer can create a new data type name for an already existing data type name. So the purpose of typedef is to redefine or rename the name of an existing data type. Syntax:

```
typedef data_type newname1, newname2,..., newnamen;
```

Example1: 

```
typedef int NUMBER;
```

Example2: 

```
typedef float SALARY;
```

The new names that are given by the user for the already existing data types are called user defined data types. In the above example NUMBER and SALARY are user defined data types.

The user defined data types can be used to declare variables as illustrated below:

Example1: 

```
typedef int NUMBER;
NUMBER N1,N2,N3;
```

Example2: 

```
typedef float SALARY;
SALARY s1,s2,s3,s4,s5;
```

##### Advantages of typedef

- User-defined names can be defined for existing data types

Example:

```
typedef int integer;
typedef char character;
integer n1, n2;
character c1;
```

From the above example, we can understand that, beginners of users of C can understand the statement *integer n1, n2* as n1 and n2 are variables of integer type and similarly, c1 is of character type.

- Based on context or application, new names can be defined for data types.

Example1: if we are developing program to represent various whole numbers then int can be renamed as below

```
typedef int WHOLE_NUMBER;
WHOLE_NUMBER w1, w2;
```

From the above two lines, we can understand that w1 and w2 will be used to represent whole numbers as the name of the data type indicates its purpose.

Example2: if we are developing program to represent percentage of three semesters of student then float can be renamed as below

```
typedef float percentage;
percentage sem_1, sem_2, sem_3;
```

From the above two lines, it can be understood that sem\_1, sem\_2 and sem\_3 represents the percentage of three semesters.

Hence, based on kind of application we are building, we can rename the existing data types as required which will be more beneficial and easy to understand.

- c. Long names of existing data types can be shortened

Example: `typedef unsigned long int ULINT;`

In the above example, the data type name unsigned long int is renamed as ULINT it is shorter than original. So instead of using long name of the data type shortened name can be used which reduce time required to type from the keyboard.

### Creating a Structure using typedef:

The structure defined using the keyword typedef is called type defined structure.

### There are two different ways to create structures using typedef

#### 1) Renaming structure type after declaration of structure using typedef.

Example:

```
struct student
{
 char name[20];
 char usn[11];
 float marks;
};
```

Once, the structure has been declared, the new data type i.e. struct student can be renamed using typedef as given below.

```
typedef struct student STUDENT;
```

Now onwards, the new name for struct student type is STUDENT. We can create variable of this structure type by using the name STUDENT.

For example: `STUDENT s;` // s is variable of type struct student

#### 2) Renaming structure type at the time of declaring structure using typedef.

The syntax of the typedefined structure along with example is as given below:

**Syntax:**

```
typedef struct [tagname]
{
 data_type1 member1;
 data_type2 member2;
 .
 .
 .
 data_typen membern;
} struct_ID ;
```



Example:

```
typedef struct student → Optional
{
 char name[20];
 char usn[11];
 float marks;

}STUDENT;
```

**NOTE:** as shown with diagram and arrow mark, the tag name i.e. student which is used to name the structure is optional.

In this example, the new data type struct student has been declared as well it is been renamed as STUDENT.

**Programming Example:** Program to simulate the addition of two complex numbers using the typedef structures is illustrated below:

```
#include<stdio.h>
typedef struct
{
 int real_part;
 int imaginary_part;
}complex;
int main ()
{
 complex c1,c2, sc;
 printf("\n Enter real and imaginary part of first complex number);
 scanf("%d %d",&c1.real_part,&c1.imaginary_part);
 printf("\n Enter real and imaginary part of second complex number);
 scanf("%d %d",&c2.real_part,&c2.imaginary_part);
 sc.real_part = c1.real_part + c2.real_part;
 sc.imaginary_part = c1.imaginary_part +c2.imaginary_part;
 printf("The real and imaginary part of resultant complex number is \n");
 printf("%d %d \n", sc.real_part, sc.imaginary_part);
 return 0;
}
```

### 5. Explain structure within a structure (Nested Structures) with examples.

A structure can be a member of another structure. A structure which includes another structure is called as nested structure. There is no limit for number of structures that can be nested.

Example:

```
struct dob
{
 int dd;
```

```
 int mm;
 int yyyy;
 };
 struct student_detail
 {
 int id;
 char name[20];
 float percentage;
 // structure within structure
 struct dob DOB;
 }stu_data;
```

In this example, “dob” structure is declared inside “student\_detail” structure. Both structure variables are normal structure variables.

To access members of the main structure and nested structure we have to use the following syntax.

To access dd of nested structure:     stu\_data.DOB.dd

To access mm of nested structure:     stu\_data.DOB.mm

To access yyyy of nested structure:   stu\_data.DOB.yyyy

This program explains how to use structure within structure in C. “student\_college\_detail” structure is declared inside “student\_detail” structure in this program. Both structure variables are normal structure variables.

Please note that members of “student\_college\_detail” structure are accessed by 2 dot(.) operator and members of “student\_detail” structure are accessed by single dot(.) operator.

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

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

```
struct student_college_detail
```

```
{
```

```
 int college_id;
```

```
 char college_name[50];
```

```
};
```

```
struct student_detail
```

```
{
```

```
 int id;
```

```
 char name[20];
```

```
 float percentage;
```

```
 // structure within structure
```

```
 struct student_college_detail clg_data;
```

```
};
```

```
int main()
```

```
{
```

```
 struct student_detail stu_data = { 1, "xyz", 90.5, 32, "REC Hulkoti"};
```

```
 printf(" Id is: %d \n", stu_data.id);
```

```
printf(" Name is: %s \n", stu_data.name);
printf(" Percentage is: %f \n\n", stu_data.percentage);
printf(" College Id is: %d \n", stu_data.clg_data.college_id);
printf("College Name is: %s \n", stu_data.clg_data.college_name);
return 0;
}
```

**6. Explain the concept of array of structures, with suitable C program.**

**Solution:**

In many situations, we need to create an array of structures. To name one example, we would use an array of students to work with a group of students stored in a structure. By putting the data in an array, we can quickly and easily work with the data.

Like arrays, Array of Structure can be initialized at compile time.

**Way1 : Initializing After Declaring Structure Array:**

struct Book

```
{
 char bname[20];
 int pages;
 char author[20];
 float price;
}b1[3] = { {"Let us C", 700, "YPK", 300.00}, {"Wings of Fire", 500, "APJ Abdul Kalam", 350.00}, {"Complete C", 1200, "Herbert Schildt", 450.00} };
```

Explanation: As soon as after declaration of structure we initialize structure with the pre-defined values. For each structure variable we specify set of values in curly braces. Suppose we have 3 Array Elements then we have to initialize each array element individually and all individual sets are combined to form single set. {"Let us C",700,"YPK",300.00}

Above set of values are used to initialize first element of the array.

Similarly - {"Wings of Fire",500,"APJ Abdul Kalam",350.00} is used to initialize second element of the array.

**Way 2: Initializing in Main**

struct Book

```
{
 char bname[20];
 int pages;
 char author[20];
 float price;
};
int main()
{
 struct Book b1[3] = { {"Let us C",700,"YPK",300.00},
 {"Wings of Fire",500,"Abdul Kalam",350.00},
 {"Complete C",1200,"Herbt Schildt",450.00}
 }
```

```
 };
 }
```

Some Observations and Important Points :

**Tip #1 : All Structure Members need not be initialized**

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

```
struct Book
```

```
{
```

```
 char bname[20];
```

```
 int pages;
```

```
 char author[20];
```

```
 float price;
```

```
}b1[3] = { {"Book1",700,"YPK"}, {"Book2",500,"AAK",350.00},
{"Book3",120,"HST",450.00} };
```

```
int main()
```

```
{
```

```
 printf("\nBook Name : %s",b1[0].bname);
```

```
 printf("\nBook Pages: %d",b1[0].pages);
```

```
 printf("\nBook Author: %s",b1[0].author);
```

```
 printf("\nBook Price : %f",b1[0].price);
```

```
}
```

Output :

Book Name : Book1

Book Pages : 700

Book Author : YPK

Book Price : 0.000000

Explanation: In this example, While initializing first element of the array we have not specified the price of book 1. It is not mandatory to provide initialization for all the values. Suppose we have 5 structure elements and we provide initial values for first two element then we cannot provide initial values to remaining elements.

```
{"Book1", 700, ,90.00}
```

above initialization is illegal and can cause compile time error.

**Tip #2 : Default Initial Value**

```
struct Book
```

```
{
```

```
 char bname[20];
```

```
 int pages;
```

```
 char author[20];
```

```
 float price;
```

```
}b1[3] = { {},{"Book2",500,"AAK",350.00}, {"Book3",120,"HST",450.00} };
```

Output :

Book Name : Book

Pages : 0

Book Author :

Book Price : 0.000000

It is clear from above output. Default values for different data types.

**C program: to illustrate array of structures.**

```
#include<stdio.h>
typedef struct
{
 char name[20]; // Name
 char usn[11]; //usn
 char subname[30]; //subject name
 int m1, m2, m3; //marks
}student;
int main()
{
 student s[100];
 int i, N;
 printf("Enter N value\n");
 scanf("%d", &N);
 //Reading student details
 printf("Enter %d student details\n", N);
 for(i=0;i<N;i++)
 {
 printf("Enter details of %d student\n", i+1);
 printf("\nEnter name of a student :");
 scanf("%s", s[i].name);
 printf("\n Enter USN:");
 scanf("%s", s[i].usn);
 printf("\n Enter subject name:");
 scanf("%s", s[i].subname);
 printf("\n Enter first, second and third IA marks :");
 scanf("%d %d %d", &s[i].m1, &s[i].m2, &s[i].m3);
 }
 // Printing student details
 printf("\n Name \t USN \t Subject Name \t Marks1 \t Marks2 \t Marks3\n");
 for(i=0;i<N;i++)
 printf("%s \t %s \t %s \t %d \t %d \t %d \n", s[i].name, s[i].usn,
s[i].subname, s[i].m1, s[i].m2, s[i].m3);
 return 0;
}
```

Explanation: in this program a structure which include name, usn, subject name and IA marks of student is declared then an array of students is declared by using student s[100] statement which defines array s with capable of storing 100 students information. Each student information is accessed by subscripting name of the array s with an index i as s[i]. In this example, N student details are inputted from the keyboard and same details are displayed on screen.

**7. Show how a structure variable is passed as a parameter to a function, with an example.**

**Solution:**

In C, structure can be passed to functions by two methods:

1. Pass by value (passing actual value of structure variable as argument)
2. Pass by reference (passing address of structure variable as argument)

**Passing structure using call-by-value:** A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

**Write a C program to create a structure student, containing name and roll number. Ask user the name and roll number of a student in main function. Pass this structure to a function and display the information in that function.**

```
#include <stdio.h>
struct student
{
 char name[50];
 int rollno;
};
void display(struct student stu); /* function prototype should be below to the
structure declaration otherwise compiler shows error */
int main()
{
 struct student s1;
 printf("Enter student's name: ");
 scanf("%s", s1.name);
 printf("Enter roll number:");
 scanf("%d", &s1.rollno);
 display(s1); // passing structure variable s1 as argument
 return 0;
}
void display(struct student stu)
{
 printf("Entered Name: %s", stu.name);
 printf("\nRoll Number: %d", stu.rollno);
}
```

**Output**

```
Enter student's name: xyz
Enter roll number: 14
Entered Name: xyz
Roll Number: 14
```

**Passing structure using call-by-reference:** The address location of structure variable is passed to a function using pass by reference technique. If structure is

passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

**Write a C program to add two distances (feet-inch system) entered by user.**

To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by pass-by-reference and display the result in main function without returning it.

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

```
struct distance
```

```
{
```

```
 int feet;
```

```
 float inch;
```

```
};
```

```
void add(struct distance d1, struct distance d2, struct distance *d3);
```

```
int main()
```

```
{
```

```
 struct distance dist1, dist2, dist3;
```

```
 printf("First distance\n");
```

```
 printf("Enter feet: ");
```

```
 scanf("%d", &dist1.feet);
```

```
 printf("Enter inch: ");
```

```
 scanf("%f", &dist1.inch);
```

```
 printf("Second distance\n");
```

```
 printf("Enter feet: ");
```

```
 scanf("%d", &dist2.feet);
```

```
 printf("Enter inch: ");
```

```
 scanf("%f", &dist2.inch);
```

```
 add(dist1, dist2, &dist3); /*passing structure variables dist1 and dist2 by
value whereas passing structure variable dist3 by reference */
```

```
 printf("\nSum of distances = %d \t %f", dist3.feet, dist3.inch);
```

```
 return 0;
```

```
}
```

```
void add(struct distance d1, struct distance d2, struct distance *d3)
```

```
{
```

```
 /* Adding distances d1 and d2 and storing it in d3 */
```

```
 d3->feet = d1.feet + d2.feet;
```

```
 d3->inch = d1.inch + d2.inch;
```

```
 if (d3->inch >= 12)
```

```
 {
```

```
 d3->inch -= 12;
```

```
 ++d3->feet;
```

```
 }
```

```
}
```

Output

First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5

Sum of distances = 18 - 2.3

Explanation: In this program, structure variables dist1 and dist2 are passed by value (because value of dist1 and dist2 does not need to be displayed in main function) and dist3 is passed by reference, i.e., address of dist3 (&dist3) is passed as an argument. Thus, the structure pointer variable d3 points to the address of dist3. If any change is made in d3 variable, effect of it is seen in dist3 variable in main function.

**8. Write a C program to store Name, USN, subject name and IA Marks of a student using structure. Print the same.**

**Program:**

```
#include<stdio.h>
struct student
{
 char name[20]; // Name
 char usn[11]; //usn
 char subname[30]; //subject name
 float m1,m2,m3; //marks
};
int main()
{
 struct student s;
 //Reading student details
 printf("\nEnter name of a student :");
 scanf("%s", s.name);
 printf("\n Enter USN:");
 scanf("%s", s.usn);
 printf("\n Enter subject name:");
 scanf("%s", s.subname);
 printf("\n Enter first, second and third IA marks :");
 scanf(" %f %f %f", &s.m1, &s.m2, &s.m3);
 // Printing student details
 printf("\n Name \t USN \t Subject Name \t Marks1 \t Marks2 \tMarks3\n");
 printf("%s \t %s \t %s \t %f \t %f \t %f \n", s.name, s.usn, s.subname, s.m1,
s.m2, s.m3);
 return 0;
}
```



9. Write a C program to store the following details of 'N' students using structure: Name: string, USN: integer, subject name: string and IA Marks: integer. Output the same details.

```
#include<stdio.h>
struct student
{
 char name[20]; // Name
 int usn; //usn
 char subname[30]; //subject name
 int m1, m2, m3; //marks
};
int main()
{
 struct student s[100];
 int i, N;
 printf("Enter N value\n");
 scanf("%d", &N);
 //Reading student details
 printf("Enter %d student details\n", N);
 for(i=0;i<N;i++)
 {
 printf("Enter details of %d student\n", i+1);
 printf("\nEnter name of a student :");
 scanf("%s", s[i].name);
 printf("\nEnter USN:");
 scanf("%d", &s[i].usn);
 printf("\nEnter subject name:");
 scanf("%s", s[i].subname);
 printf("\nEnter first, second and third IA marks :");
 scanf(" %d %d %d", &s[i].m1, &s[i].m2, &s[i].m3);
 }
 // Printing student details
 printf("\n Name \t USN \t Subject Name \t Marks1 \t Marks2 \tMarks3\n");
 for(i=0;i<N;i++)
 printf("%s \t %d \t %s \t %d \t %d \t %d \n", s[i].name, s[i].usn,
s[i].subname, s[i].m1, s[i].m2, s[i].m3);
 return 0;
}
```

10. Write a C program to input the following details of 'N' students using structure:

Roll No: integer, Name: string, Marks: float, Grade: char

Print the names of the students with marks $\geq$ 70.0%.

Program:

```
#include<stdio.h>
struct student
{
 int rno; // Roll No
 char name[20]; // Name
 float marks; //Marks
 char grade; //Grade
};
int main()
{
 int i, n, found=0;
 struct student s[20];
 printf("\nHow many student details ?");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 printf("\n Type in %d student detail \n",i+1);
 printf("\n Roll no :");
 scanf("%d" , &s[i].rno);
 printf("\n Name:");
 scanf("%s" , s[i].name);
 printf("\n Marks:");
 scanf("%f" , &s[i].marks);
 printf("\n Grade :");
 scanf(" %c" , &s[i].grade);
 }
 printf("\n The Names of the students with marks >=70.0%\n");
 for(i=0;i<n;i++)
 {
 if(s[i].marks>=70.0)
 {
 printf("\n %s\n" , s[i].name);
 found =1;
 }
 }
 if(found ==0)
 printf("\n There is no student with marks >=70.0%\n");
 return 0;
}
```

**11. Write a C program to maintain a record of “n” student details using an array of structures with four fields (Roll number, Name, Marks, and Grade). Each field is of an appropriate data type. Print the marks of the student given student name as input.**

**Program:**

```
#include<stdio.h>
#include<string.h>
struct student
{
 int rno; // Roll No
 char name[20]; // Name
 float marks; //Marks
 char grade; //Grade
};
int main()
{
 int i, N;
 struct student s[20];
 char key[20];
 printf("\nHow many student details ?");
 scanf("%d",&N);
 for(i=0;i<N;i++)
 {
 printf("\n Type in %d student detail \n",i+1);
 printf("\n Roll no :");
 scanf("%d" , &s[i].rno);
 printf("\n Name:");
 scanf("%s" , s[i].name);
 printf("\n Marks:");
 scanf("%f" , &s[i].marks);
 printf("\n Grade :");
 scanf("%c" , &s[i].grade);
 }
 printf("Enter the name to be searched\n");
 scanf("%s", key);
 for(i=0; i<N; i++)
 {
 if(strcmp(key, s[i].name))
 {
 printf("\n Marks obtained by %s = %d\n" , key, s[i].marks);
 return 0;
 }
 }
 printf("\n No student found with name = %s \n", key);
}
```

```
 return 0;
}
```

## File Handling

### 12. What is a file? What are the different modes of opening a file? Explain.

**Solution:** Abstractly, a file is a collection of data stored on a secondary storage device, which is generally a disk. The collection of data may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image.

Files can be classified as input files and output files. Input files are file which contains various data that will be provided as input to the program. Output files are files which contains the output produced from the program.

**Working with file:** While working with file, you need to declare a pointer of type FILE. This declaration is needed for communication between file and program.

```
FILE *ptr;
```

Here, ptr is pointer to file called as file pointer which will be used to address or reference the file.

### Different Modes of file opening in Standard I/O

In C, the files are opened using fopen function. Which accepts two parameters; first parameter is name of the file to be opened and second parameter is mode of file open.

General Syntax of fopen call:

```
FILE *fopen(char filename[], char mode[]);
```

On successful opening of specified file, fopen function returns a file pointer. On failure, fopen function returns NULL.

| File Access Mode | Meaning of Mode                                       | If file doesn't exist                                                                             |
|------------------|-------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| "r"              | Open for reading                                      | If the file does not exist, fopen() returns NULL.                                                 |
| "w"              | Open for writing                                      | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| "a"              | Open for appending i.e data is added from end of file | If the file doesn't exist it will be created                                                      |
| "r+"             | Open for both reading and writing                     | If the file does not exist, fopen() returns NULL.                                                 |
| "w+"             | Open for both reading and writing                     | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| "a+"             | Open for both reading and appending                   | If the file does not exists, it will be created.                                                  |

**13. Explain how the input is accepted from a file and displayed.****Solution:**

**Accepting input from a file and displaying can be done in different ways.**

**a. Using fscanf() and fprintf()**

**fscanf** function is used to accept input from a file. Which accepts three parameters; first parameter is source of input which can be either file or standard input device, second parameter defines the type of data to read and third parameter defines address of variable where the data is to be stored. Once the data is stored in variables then it can be displayed on standard output device i.e., monitor or can be stored in a file using fprintf().

This program reads a string of text from a file using fscanf() and fprintf().

**/\* Source Code to read a text of string and display the same from a file. \*/**

```
#include <stdio.h>
#include<stdlib.h> // for exit function
int main()
{
 char data[1000];
 FILE *fptr;
 if ((fptr=fopen("program.txt","r"))==NULL)
 {
 printf("Error! opening file");
 exit(1); /* Program exits if file pointer returns NULL. */
 }
 fscanf(fptr,"%[^\\n]",data); //read every character that is not '\\n'
 fprintf(stdout,"Data from file:\\n%s",data);
 fclose(fptr);
 return 0;
}
```

This program reads the content of “program.txt” file and stores it in a string data. Then content of this data is displayed on screen using fprintf(). If there is no file named program.txt then, this program displays Error! opening file and program will be terminated.

**b. Using fgetc and fputc function**

**fgetc()** function is used to read one character at a time from a specified file. This procedure can be repeated until end of file.

**fputc()** function is used to write one character at a time into a specified file. This procedure can be repeated until all the characters are written to a file.

**/\* program to read text from file and store it in another file using fgetc and fputc \*/**

```
#include <stdio.h>
#include<stdlib.h> // for exit function
int main()
{
 char ch;
```

```

FILE *fptr1,*fptr2;
if ((fptr1=fopen("read.txt","r"))==NULL)
{
 printf("Error! opening file");
 exit(1); /* Program exits if file pointer returns NULL. */
}
if ((fptr2=fopen("write.txt","w"))==NULL)
{
 printf("Error! opening file");
 exit(1); /* Program exits if file pointer returns NULL. */
}
while((ch = fgetc(fptr1)) != EOF) // Read a Character until end of file
 fputc(ch,fptr2);
fclose(fptr1);
fclose(fptr2);
return 0;
}

```

### c. Using fgets and fputs functions

These are useful for reading and writing entire lines of data to/from a file. If *buffer* is a pointer to a character array and *n* is the maximum number of characters to be stored, then

*fgets (buffer, n, input\_file);*

will read an entire line of text (max chars = n) into *buffer* until the newline character or n=max, whichever occurs first. The function places a NULL character after the last character in the buffer. The fgets function returns NULL when no more data to read.

*fputs (buffer, output\_file);*

writes the characters in *buffer* until a NULL is found. The NULL character is not written to the *output\_file*.

NOTE: fgets does not store the newline into the buffer, fputs will append a newline to the line written to the output file.

**/\* program to read text from file and store it in another file using fgets and fputs \*/**

```

#include <stdio.h>
#include<stdlib.h> // for exit function
int main()
{
 char data[101];
 FILE *fptr1,*fptr2;
 if ((fptr1=fopen("read.txt","r"))==NULL)
 {
 printf("Error! opening file");
 exit(1); /* Program exits if file pointer returns NULL. */
 }
 if ((fptr2=fopen("write.txt","w"))==NULL)

```

```

 {
 printf("Error! opening file");
 exit(1); /* Program exits if file pointer returns NULL. */
 }
 /* Read 100 Characters or until end of file whichever occurs first from read.txt
file and write it to write.txt file. */
 while((fgets(data,100,fptr1)) != NULL)
 fputs(data,fptr2);
 fclose(fptr1);
 fclose(fptr2);
 return 0;
}

```

#### 14. Explain the following file operations along with syntax and examples:

a) **fopen()**      b) **fclose()**      c) **fscanf()**      d) **fprintf()**      e) **feof()**

##### Solution:

a) **fopen()** : Opening a file is done by a call to the function `fopen( )` which tells the operating system the name of the file and whether the file is to be opened for reading or for writing or for appending.

General form of call to `fopen`:

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

OR

```
filepointer = fopen ("Filename" , "mode");
```

The function `fopen` takes two parameters both of which are strings.

The parameter **filename** is a C string which must contain the name of the disk file to be opened. If the file is not in the default directory a full path name must be provided.

Parameter **mode** is a C string which defines the way the file is to be opened. The mode is a string not characters and must be enclosed in double quotation marks.

The various modes are shown below.

| Access Mode | Description                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "r"         | Open a file for <b>reading</b> . Specified file must exist.                                                                                                                                                                                        |
| "w"         | Creates an empty file for <b>writing</b> . If the specified file doesn't exist then, it creates a new file by specified name and opens for writing. If the file already exist then, its content is erased and opens as new empty file for writing. |
| "a"         | Open a file for <b>appending</b> . If the file doesn't exist then, it creates new file by specified name and opens it for appending.                                                                                                               |
| "r+"        | Open a file for both reading and writing. Specified file must exist.                                                                                                                                                                               |
| "w+"        | Creates an empty file for both reading and writing.                                                                                                                                                                                                |
| "a+"        | Opens a file for both reading and appending.                                                                                                                                                                                                       |

#### Example1: Creating and opening three different files in different modes

```
FILE *fp1,*fp2,*fp3;
```

```
fp1 = fopen("student.txt","r"); // opens a file for reading
fp2 = fopen("One.txt","w"); // opens a file for writing
fp3 = fopen("newfile.txt","a"); // opens a file for appending
```

Note:

1. One must include the complete path of the file if the file is not stored in the default directory.

Example:

```
fp1 = fopen("C:\\foldername\\xyz.txt", "r");
```

2. The data file may be named with extension .dat or .txt

ii) **fclose()** : After file has been used it must be closed. This is done by a call to the function `fclose()`. The `fclose()` function breaks the connection between the stream and the file.

General form of a call to the `fclose()` is

```
fclose(file_pointer);
```

The function `fclose` takes one parameter which is pointer to a file of opened file. If there is an error, such as trying to close a file that is not opened, the function returns EOF; otherwise it returns 0. The return value is usually not checked.

Example :

```
File *fp1,*fp2 ;
fclose(fp1) ; // closes a file referenced by fp1
fclose(fp2); // closes a file referenced by fp2
```

### Opening and closing the File : Example program

```
int main()
{
 FILE *fp;
 char ch;
 fp = fopen("INPUT.txt","r"); // Open file in Read mode
 while(1)
 {
 ch = fgetc(fp); // Read a Character
 if(ch == EOF) // Check for End of File
 break ;
 printf("%c",ch);
 }
 fclose(fp); // Close File after Reading
}
```

### iii) **fscanf()**

The C library function `fscanf()` reads formatted input from a file or from the standard input device i.e. keyboard.

General syntax of `fscanf()` is

```
fscanf(filepointer, "format string", list of address of variables);
```



fscanf() returns number of data items successfully read.

Here,

- **filepointer** – This is the pointer to a FILE object that identifies the opened file or it can be **stdin** if we want to read data from keyboard.
- **format string** – This is the C string that contains one or more format specifiers which defines type of data to read from the file or from the keyboard.
- **List of address of variables** – this contains one or more address of variables where the data will be stored.

// Example program which reads the data from file using fscanf()

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
 char str1[10], str2[10], str3[10];
```

```
 int year;
```

```
 FILE * fp;
```

```
 fp = fopen ("file.txt", "w+");
```

```
 fputs("We are in 2012", fp);
```

```
 rewind(fp);
```

```
 fscanf(fp, "%s %s %s %d", str1, str2, str3, &year); // reads three strings and
one integer data
```

```
 printf("Read String1 |%s|\n", str1);
```

```
 printf("Read String2 |%s|\n", str2);
```

```
 printf("Read String3 |%s|\n", str3);
```

```
 printf("Read Integer |%d|\n", year);
```

```
 fclose(fp);
```

```
 return(0);
```

```
}
```

#### iv) **fprintf()**

The C library function fprintf() prints formatted output to a file, printer or to the standard output device i.e. screen.

General syntax of fprintf() is

```
fprintf(filepointer, "format string", list of variables);
```

fprintf() returns number of data items successfully written.

Here,

- **filepointer** – This is the pointer to a FILE object that identifies the opened file or it can be **stdout** if we want to write data to screen.
- **format string** – This is the C string that contains one or more format specifiers which defines type of data to be written to the file or standard output device.

- **List of variables** – this contains one or more variables of which the data will be written.

```
/* example program to write data to a file */
#include <stdio.h>
#include <stdlib.h>
int main()
{
 FILE * fp;
 fp = fopen ("file.txt", "w+");
 fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012); // writes three
strings followed by an integer separated by whitespace to the file file.txt
 fclose(fp);
 return(0);
}
```

#### v) **feof()**

The C library function **int feof(FILE \*filepointer)** tests the end-of-file indicator for the given file. Where, filepointer is the pointer to file object which identifies the file to be tested.

This function returns a non-zero value when End-of-File is encountered, else zero is returned.

// use of feof() in program

```
#include <stdio.h>
int main ()
{
 FILE *fp;
 int c;
 if((fp = fopen("file.txt", "r"))==NULL)
 {
 printf("Error in opening file");
 return (-1);
 }
 while(!feof(fp)) //feof() tests the End-of-File, if EOF then jumps out of loop.
 {
 c = fgetc(fp);
 printf("%c", c);
 }
 fclose(fp);
 return(0);
}
```

#### 15. Explain the following file operations along with syntax and examples:

- a) **fputc()**    b) **fputs()**    c) **fgetc()**    d) **fgets()**

**a) fputc()**

The C library function **int fputc(int character, FILE \*filepointer)** writes a character (an unsigned char) specified by the argument **char** to the specified file and advances the position indicator for the file.

Here,

- **character** - This is the character to be written. This is passed as its int promotion.
- **filepointer** -- This is the pointer to a FILE object that identifies the file where the character is to be written.

If there are no errors, the same character that has been written is returned. If an error occurs, EOF is returned and the error indicator is set.

**// example program to illustrate use of fputc()**

```
#include <stdio.h>
int main ()
{
 FILE *fp;
 int ch;
 fp = fopen("file.txt", "w+");
 for(ch = 'A' ; ch <= 'Z'; ch++)
 fputc(ch, fp); // writes a character stored in ch to file pointed by fp
 fclose(fp);
 return(0);
}
```

**b) fputs()**

The C library function **int fputs(const char \*str, FILE \*filepointer)** writes a string to the specified file up to but not including the null character.

Here,

- **str** - This is an array containing the null-terminated sequence of characters to be written.
- **filepointer** - This is the pointer to a FILE object that identifies the file where the string is to be written.

This function returns a non-negative value on success or EOF on error.

**// program to illustrate use of fputs()**

```
#include <stdio.h>
int main ()
{
 FILE *fp;
 fp = fopen("file.txt", "w+");
 //writes this is c programming to file pointed by fp
 fputs("This is c programming.", fp);
 //writes This is a system programming language. to the file pointed by fp.
 fputs("This is a system programming language.", fp);
 fclose(fp);
}
```

```
 return 0;
}
```

### c) **fgetc()**

The C library function **int fgetc(FILE \*filepointer)** gets the next character (an unsigned char) from the specified file and advances the position indicator for the file.

Where,

- **filepointer** - This is the pointer to a FILE object that identifies the file on which the operation is to be performed.

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

```
// program to illustrate use of fgetc()
#include <stdio.h>
int main ()
{
 FILE *fp;
 int c;
 int n = 0;
 fp = fopen("file.txt", "r");
 if(fp == NULL)
 {
 printf("Error in opening file");
 return (-1);
 }
 while(!feof(fp)) //read until end of file
 {
 c = fgetc(fp); // reads an unsigned character and converts to integer.
 printf("%c", c);
 }
 fclose(fp);
 return(0);
}
```

### d) **fgets()**

The C library function **char \*fgets(char \*str, int n, FILE \*filepointer)** reads a line from the specified stream and stores it into the string pointed to by **str**.

It stops when either **n-1** characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

Where,

- **str** -- This is the pointer to an array of characters where the string read is stored.

- **n** -- This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array is passed.
- **filepointer** -- This is the pointer to a FILE object that identifies the file where characters are read from.

On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned.

If an error occurs, a null pointer is returned.

// example program to illustrate use of fgets()

```
#include <stdio.h>
int main()
{
 FILE *fp;
 char str[60];
 /* opening file for reading */
 fp = fopen("file.txt" , "r");
 if(fp == NULL)
 {
 printf("Error opening file");
 return (-1);
 }
 // fgets() reads maximum of 60 characters from file pointed by fp and
 stores in str
 if(fgets (str, 60, fp)!=NULL)
 {
 /* writing content to stdout */
 puts(str);
 }
 fclose(fp);
 return(0);
}
```

#### 16. Write a C program to read and display a text from the file.

/\* Source Code to read a text of string and display the same from a file. \*/

```
#include <stdio.h>
#include<stdlib.h> // for exit function
int main()
{
 char ch;
 FILE *fptr;
 if ((fptr=fopen("program.txt","r"))==NULL)
 {
 printf("Error! opening file");
```

```
 exit(1); /* Program exits if file pointer returns NULL. */
 }
 printf("Content of file is:\n");
 while((ch=fgetc(fptr))!=EOF)
 fputc(ch, stdout);
 fclose(fptr);
 return 0;
}
```

This program reads the content of “program.txt” file one character at a time using fgetc function and stores it in a variable ch. Then content of this variable ch is displayed on screen using fputc(). If there is no file named program.txt then, this program displays Error! opening file and program will be terminated.

**16. Write a C program to read the contents from the file called ‘abc.txt’, count the number of characters, number of lines and number of whitespaces and output the same.**

**Program:**

```
#include<stdio.h>
int main()
{
 FILE *fp;
 char ch;
 int cc=0; /* number of characters */
 int bc=0; /* number of blanks */
 int tc=0; /* number of tabs */
 int lc=0; /* number of lines */
 int wc=0; /* number of words */
 fp = fopen("abc.txt", "r");
 if(fp==NULL)
 {
 printf("Error in opening the file \n");
 return -1;
 }
 while((ch=fgetc(fp))!=EOF)
 {
 cc++;
 if (ch == " ")
 bc++;
 if(ch == "\n")
 lc++;
 if(ch == "\t")
 tc++;
 }
 fclose(fp);
}
```

```
 wc = bc + lc;
 printf("\n Number of characters = %d\n", cc);
 printf("\n Number of tabs = %d\n", tc);
 printf("\n Number of lines = %d\n", lc);
 printf("\n Number of blanks = %d\n", bc);
 printf("\n Number of words count = %d\n", wc);
 return 0;
}
```

- 17. Given two text documentary files “Ramayana.in” and “Mahabharatha.in”. Write a C program to create a new file “Karnataka.in” that appends the content of the file “Ramayana.in” to the file “Mahabharatha.in”. Also calculate the number of words and new lines in the output file.**

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 char buff;
 FILE *fd1,*fd2,*fd3;
 int line=0,word=0;
 fd1=fopen("Mahabharatha.in","r");
 if(fd1==NULL)
 {
 printf("Error: opening Mahabharatha.in file");
 exit(1);
 }
 fd2=fopen("Ramayana.in","r");
 if(fd2==NULL)
 {
 printf("Error: opening Ramayana.in file");
 exit(1);
 }
 fd3=fopen("Karnataka.in","w");
 if(fd3==NULL)
 {
 printf("Error: Opening Karnataka.in file");
 exit(1);
 }
 //write the content of Mahabharatha.in to the file Karnataka.in
 while((buff=fgetc(fd1))!=EOF)
 fputc(buff,fd3); //write into the file
 //Append the content of Ramayana.in to the file Karnataka.in
 while((buff=fgetc(fd2))!=EOF)
 fputc(buff,fd3);
}
```

```
fclose(fd1);
fclose(fd2);
fclose(fd3);
printf("Content of Karnataka.in file is:\n");
fd1=fopen("Karnataka.in","r");
while((buff=fgetc(fd1))!=EOF)
{
 if(buff=='\n')
 line++;
 if(isspace(buff)||buff=='\t'||buff=='\n')
 word++;
 putchar(buff);
}
fclose(fd1);
printf("\n no of lines=%d\n",line);
printf("no of words=%d\n",word);
return 0;
}
```

18. Given two university information files “studentname.txt” and “usn.txt” that contains students Name and USN respectively. Write a C program to create a new file called “output.txt” and copy the content of files “studentname.txt” and “usn.txt” into output file in the sequence shown below. Display the contents of output file “output.txt” on to the screen.

| Student Name | USN  |
|--------------|------|
| Name1        | USN1 |
| Name2        | USN2 |
| ...          | ...  |
| ...          | ...  |

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 char buff;
 FILE *fd1,*fd2,*fd3;
 fd1=fopen("USN.txt","r");
 if(fd1==NULL)
 {
 printf("Error: opening USN.txt file");
 exit(1);
 }
 fd2=fopen("OUTPUT.txt","w");
 if(fd2==NULL)
```



```
{
 printf("Error: Opening OUTPUT.txt file");
 exit(1);
}
fd3=fopen("studentname.txt","r");
if(fd3==NULL)
{
 printf("Error: opening studentname.txt file");
 exit(1);
}
fprintf(fd2,"%s\t%s\n","Student Name","USN");
do
{
 buff=fgetc(fd3); //read content from file studentname.txt
 if((buff!=EOF) && (buff!='\n'))
 fputc(buff,fd2); //write into the file output.txt
 if(buff=='\n')
 {
 fputc('\t',fd2);
 do
 {
 buff=fgetc(fd1); //read content from USN.txt file
 if(buff!=EOF)
 fputc(buff,fd2); //write into the file output.txt
 if(buff=='\n')
 break;
 }while(1); //until end-of-file for USN.txt
 }
}while(buff!=EOF); //until end-of-file for NAME.txt
fclose(fd1);
fclose(fd2);
fclose(fd3);
printf("Content of output.txt file is:\n");
fd1=fopen("output.txt","r");
while((buff=fgetc(fd1))!=EOF)
 putchar(buff);
fclose(fd1);
return 0;
}
```

## MODULE 5: Pointers, Preprocessor Directives and Data Structures

### 1. What is pointer? Explain with an example program.

**Solution:**

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

Two important operators used with pointers are

- Ampersand operator ( & )** – it is used to access address of a variable
- Dereferencing or Indirection Operator ( \* )** – it is used to declare pointer and access value of a variable referenced by pointer. The process of accessing value of a variable indirectly by pointer is called as indirection.

Consider the following Program to illustrate pointers and their usage.

```
#include<stdio.h>
int main()
{
 int var; //declares an integer variable var
 int *ptr; // declares a pointer to integer variable
 ptr = &var; // initializes pointer ptr to address of variable var
 var = 10; // stores 10 into variable var
 //prints value of variable i.e. 10
 printf("Value of Variable = %d\n", var);
 // *ptr accesses value of variable var referenced by pointer ptr hence, it is again 10
 printf("value of variable using pointer = %d\n", *ptr);
 //&var retrieves address of variable var
 printf("Address of var = %u\n", &var); // it prints address of var
 // ptr contains address of variable var
 printf("Address of var using pointer = %u\n", ptr); // it prints address of var
 // modify the value of variable var using pointer
 *ptr = *ptr * 5; // *ptr = 10 * 5 = 50, hence it stores 50 in variable var
 printf("New value of var = %d\n", *ptr); // prints 50
 return 0;
}
```

### 2. Define pointer? With example, explain declaration and initialization of pointer variables.

**Solution:**

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

Two important operators used with pointers are

- Ampersand operator ( & )** – it is used to access address of a variable
- Dereferencing or Indirection Operator ( \* )** – it is used to declare pointer and access value of a variable referenced by pointer. The process of accessing value of a variable indirectly by pointer is called as indirection.

**Declaration of pointer:**

Every pointer variable before it is used, it must be declared. Declaration tells the compiler the name of pointer, to which type of variable it points.

#### General Syntax of Pointer Declaration:

Datatype          \*pointer\_name;

Where, pointer\_name should be framed with the help of rule of identifier.

Datatype can be any valid C data type to which the pointer will point.

Ex:

```
int *ptr; // ptr is a pointer to integer
char *c; // c is pointer to character
float *fp; // fp is a pointer to floating point variable
```

#### Initialization of Pointer variable:

The process of assigning address of a variable to the pointer variable is called as initialization.

#### General Syntax of Initialization:

pointer\_name = &variable;

Here, the address of variable is assigned to pointer.

Example:

```
int var;
```

After declaration of variable var, it is allocated in some memory location (address) say 1000.

```
int *p;
```

here, it declares pointer p which can contain the address of any integer variable.

```
p = &var;
```

here, the address of variable var i.e 1000 is stored into pointer variable p. Therefore p contains 1000. Now onwards p acts as a pointer to var.

**r-value:** if \*pointer\_name is used on the right side of an assignment operator(=) in an expression then, it refers to the value of a variable referred by pointer. It is called as read-value.

**l-value:** if \*pointer\_name is used on the left side of an assignment operator(=) in an expression then, it refers to the address of a variable referred by pointer. It is called as location-value.

### 3. Explain functions and pointers with example.

Like variables, pointer variables can also be used as parameters in the functions. This mechanism is known as call-by-address or call-by-reference mechanism.

In this call-by-address mechanism, the address of actual parameter are copied to the formal parameters. Any modification to the formal parameters in the function definition will affect actual parameters.

For Example:

```
Datatype *function_name(datatype *ptr1, datatype *ptr2, ...)
{
 //set of statements
```

}

In this example, the function accepts address of actual parameters and returns address of result obtained.

Example Program: To swap two numbers using pointers and functions

// Pointer is a variable which contains address of another variable

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

```
void swap(int *x, int *y);
```

```
int main()
```

```
{
```

```
 int p, q;
```

```
 printf("Enter the value for p and q\n");
```

```
 scanf("%d%d", &p, &q);
```

```
 printf("In Main: Before Function Call\n");
```

```
 printf("p=%d, q=%d\n", p, q);
```

```
 swap(1000,2500)
```

```
 swap(&p,&q);
```

```
 printf("In Main: After Function Call\n");
```

```
 printf("p=%d, q=%d\n", p, q);
```

```
 return 0;
```

```
}
```

```
void swap(int *x, int *y)←
```

```
{
```

```
 int temp;
```

```
 printf("In Swap: Before Exchange\n");
```

```
 printf("x=%d, y=%d\n", *x, *y);
```

```
 temp=*x;
```

```
 *x=*y;
```

```
 *y=temp;
```

```
 printf("In Swap: After Exchange\n");
```

```
 printf("x=%d, y=%d\n", *x, *y);
```

```
}
```

Copies address of p (&p) assumed that it is at address 1000 to formal parameter \*x and address of q (&q) assumed that it is at address 2500 to formal parameter \*y and transfers control to function definition

**Output of the above program is**

Enter the value for p and q

10     20

**In Main: Before Function Call**

**p=10, q=20**

In Swap: Before Exchange

x=10, y=20

In Swap: After Exchange

x=20, y=10

**In Main: After Function Call**

**p=20, q=10**

From the above output and flow shown with arrow and text description, we can understand that the address of actual parameters p and q are copied to formal parameters \*x and \*y respectively. From the bold text of output we can understand that the modification that we have done for formal parameters in function definition have affected(modified) actual parameters in calling function hence, the p and q value before function call and after function are different.

#### 4. What are preprocessor directives? Explain different types.

**Solution:**

**Preprocessor Directives:** Preprocessor directives are the statements of C programming language which begins with pound (#) symbol. They are used in the program to instruct or tell the compiler to perform the following

- including external file
- defining the constants or symbols or macros and
- controlling the execution of statements conditionally or unconditionally.

Examples : #define , #include, #if, #endif , #ifdef and #else

**Three types of pre-processor directives are:**

- a. **Macro Substitution Directives:** It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. This is done by using #define statement. This statement usually known as a macro definition (or simply a macro) and it takes the following general form :

#define identifier string

Where, identifier is a name of a macro, symbol or constant and string is one or more tokens.

Example :

```
#define COUNT 100 // defines COUNT with value 100
#define FALSE 0 // defines FALSE with value 0
#define PI 3.142 // defines PI with value 3.142
```

- b. **File Inclusion Directives:** An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive #include.

**General form:**

#include<filename> OR #include "filepath"

Where, filename and filepath refers to the file to included in the program.

Example:

```
#include<stdio.h> // includes stdio.h header file
#include<stdlib.h> // includes stdlib.h header file
#include "TEST.C" // includes TEST.c program file
#include "SYNTAX.C" // includes SYNTAX.C program file
```

- c. **Compiler Control Directives:** C preprocessor offers a feature known as conditional compilation which can be used to control a particular line or group of lines in a program.

**Various directives used to control compiler execution are**

#if - can be used for two-way selection  
 #ifdef - if defined  
 #ifndef - if not defined  
 #elif - else if directive  
 #else - else  
 #endif - end of if directive

**Example 1: To ensure that a constant is defined in the program**

```
#include<define.h>
#ifndef TEST
#define TEST 1
#endif
```

In this example, #ifndef directive checks whether a constant TEST is defined or not. If not defined then it defines TEST with value 1 otherwise it does nothing.

**Example 2: To ensure that selected part of program is executed**

```
#include<define.h>
#if num < 10
// execute part A section
#else
// execute part B section
#endif
```

In this example, #if directive tests the condition num<10 is true or false. If true, then it allows the compiler to execute part A section code otherwise it executes part B section code.

**5. What is static and dynamic memory allocation? Give the differences between the same. Explain dynamic memory allocation functions.**

**Solution:**

**Static memory allocation:** the process of allocating memory during compile time is known as static memory allocation.

**Dynamic memory allocation:** the process of allocating memory during run-time or execution time as required is known as dynamic memory allocation.

**Difference between static and dynamic memory allocation**

| Static memory allocation                        | Dynamic memory allocation                 |
|-------------------------------------------------|-------------------------------------------|
| Memory is allocated during compile time         | Memory is allocated during run-time       |
| Amount of memory allocated is fixed or constant | Amount of memory allocated can be changed |
| Wastage of memory may occur                     | No wastage of memory                      |
| Memory is allocated from stack segment          | Memory is allocated from heap segment     |

**Dynamic memory allocation functions:**

The following functions are used to allocate and deallocate the memory from heap segment dynamically. Here, the first three functions are used to allocate memory and last one is used to deallocate the memory. These functions are defined in header file alloc.h or stdlib.h.

- a. malloc()
- b. calloc()
- c. realloc()
- d. free()

- a. **malloc() function:** It is used to allocate a single block of memory of specified size dynamically and returns a pointer to the allocated block. The initial values in all the memory locations will be garbage values.

The general form for memory allocation using malloc is:

```
datatype *ptr =(data type *)malloc (requiredAmountofmemory);
```

**Example 1 :**

```
int *ptr;
ptr = (int *) malloc(10*sizeof(int));
```

In this example, it allocates  $10 * \text{sizeof(int)} = 10 * 4 \text{ bytes} = 40 \text{ bytes}$  of memory and starting address of block is assigned to pointer ptr.

**Example 2 :**

```
char *ptr ;
ptr = (char *) malloc(5*sizeof(char));
```

In this example, it allocates  $5 * \text{sizeof(char)} = 5 * 1 \text{ bytes} = 5 \text{ bytes}$  of memory and starting address of block is assigned to pointer ptr.

- b. **calloc() function:** It is used to allocate multiple blocks of memory of specified size dynamically and returns a pointer to the allocated block. The initial values in all the memory locations will be ZERO.

The general form for memory allocation using calloc is:

```
datatype *ptr =(data type *)calloc (NoOfBlocks , requiredAmountofmemory);
```

**Example 1 :**

```
int *ptr;
ptr = (int *) calloc(10, sizeof(int));
```

In this example, it allocates 10blocks of memory. Each block will be of size 4bytes since size of int is 4bytes in 32bit machine. Therefore, total of 40 bytes of memory is allocated and starting address of first block is assigned to pointer ptr.

**Example 2 :**

```
char *ptr ;
ptr = (char *) calloc(5, sizeof(char));
```

In this example, it allocates 5blocks of memory. Each block will be size 1byte since size of char is 1byte. Therefore, total of 5bytes of memory is allocated and starting address of first block is assigned to pointer ptr.

- c. **realloc() function:** It is used to increase or decrease the size of already allocated memory using malloc or calloc function and returns a pointer to the newly allocated block.

The general form for memory allocation using realloc is:

```
datatype *ptr =(data type *)realloc (ptrname, NewAmountofmemory);
```

**Example 1 :**

```
int *ptr;
ptr = (int *) malloc(10*sizeof(int));
```

In this example, it allocates  $10 * \text{sizeof}(\text{int}) = 10 * 4 \text{ bytes} = 40 \text{ bytes}$  of memory and starting address of block is assigned to pointer ptr.

```
ptr = (int *)realloc(ptr, 20);
```

in this example, already allocated memory of 40bytes pointed by ptr is decreased to 20bytes.

**Example 2 :**

```
char *ptr ;
```

```
ptr = (char *) malloc(5*sizeof(char));
```

In this example, it allocates  $5 * \text{sizeof}(\text{char}) = 5 * 1 \text{ bytes} = 5 \text{ bytes}$  of memory and starting address of block is assigned to pointer ptr.

```
ptr = (char *) realloc(ptr, 10*sizeof(char));
```

In this example, already allocated memory of 5 bytes pointed by ptr is increased to 10bytes using realloc.

- d. **free()** – it is used to deallocate or release the memory which was already allocated by using malloc, calloc or realloc function.

The general form using free is:

```
free(pointer_name);
```

where, pointer\_name must be pointing to block of memory allocated using malloc, calloc or realloc function.

**Example 1:**

```
char *ptr ;
```

```
ptr = (char *) malloc(5*sizeof(char));
```

```
free(ptr); // releases memory pointed by ptr
```

**Example 2:**

```
char *p ;
```

```
p = (int *) malloc(5*sizeof(int));
```

```
free(p); // releases memory pointed by p
```

## 6. What are primitive and non-primitive data types? Explain.

**Solution:**

**Primitive and Non Primitive Data Types:** Data is recorded facts and figures that can be retrieved and manipulated in order to produce useful information / results. A data type is a type of value of the variable that is stored in particular memory location.

Data Types can be broadly classified into Primitive and Non Primitive Data Types

**Primitive Data Types:** The primitive data types are basic or fundamental data types that can be manipulated or operated directly by machine instructions.

The C language provides the following primitive data types:

- char data type – it is used to define characters
- int data type – it is used to define integer values
- float data type – it is used to define single precision floating point numbers
- double – it is used to define double precision floating point numbers
- void – it is empty data type and used as generic data type.



**Non Primitive Data Types:** are those that are not defined by the programming language but are instead created by the programmer. These are also known as derived data types which are derived from the existing fundamental data types.

Non Primitive Data Types are classified into two types

a. **Linear Data Types**

b. **Non Linear Data types**

a. **Linear Data Types:** Here the data elements are arranged in linear fashion.

Examples:

- **stack** – it is a linear data structure, where insertion and deletion are done from only one end i.e top end. It is also known as Last-In-First-Out(LIFO) data structure.
- **Queue** – it is a linear data structure, where insertion is done from rear end and deletion is done from front end. It is also known as First-In-First-Out(FIFO) data structure.
- **Linked list** – it is a linear data structure where insertion and deletion are done linearly.

b. **Non Linear Data Types:** Here the data elements are arranged in nonlinear fashion.

Examples:

- **Trees** – it is a non-linear data structure, where data are arranged in non-linear fashion.
- **Graphs** – where set of vertices and edges are arranged non-linearly.
- **Maps** - where different states, districts, cities, etc are represented in non-linear fashion.

## 7. What is abstract data type? Explain with examples.

### Solution:

Abstract data type (ADT) is a process of defining data items and various operations which can be performed on data items in abstract form. It hides the implementation details of how the data items are stored in memory and how the operations are actually implemented.

Example 1: Define ADT for stack

ADT stack

Data items: top, array

Operations: push(), pop(), display(), stackOverflow(), stackUnderflow()

End ADT

Example 2: Define ADT for Queue

ADT Queue

Data items: rear, front, array

Operations: insert\_rear(), delete\_front(), display(), queueOverflow(), queueUnderflow()

End ADT

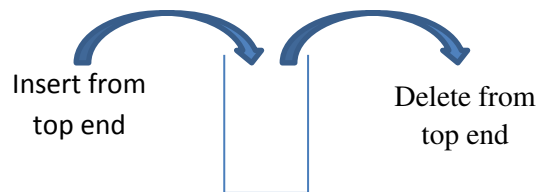
In the above example, we can observe that data items and various operations are listed but not detailed any information about how the data items are stored in memory, manipulated and operated. Similarly, how the operations are implemented is also not described. Hence, it hides the implementation details from the user.

## 8. Explain the following

- a. Stack      b. Queue      c. Linked List      d. Tree

a. **Stack:** It is a linear data structure, where insertion and deletion are done from only one end i.e., top end. It is also known as **Last-In-First-Out (LIFO)** data structure.

**Pictorial View of Stack:**



**Various Applications of Stack:**

1. It is used to convert infix expression into postfix expression
2. It is used to evaluate postfix expression
3. It is used to store activation records during recursion in system
4. It is used to reverse string or numbers
5. It is used to store browsers data or history

**Various Operations of Stack:**

1. **Push():** this function is used to insert a data item from top of the stack.
2. **Pop():** this function is used to delete a data item from top of the stack.
3. **Display():** this function is used to list or print all the data items in the stack.
4. **stackOverflow():** this function is used to check whether stack is full or not.
5. **stackUnderflow():** this function is used to check whether stack is empty or not.

b. **Queue:** it is a linear data structure, where insertion is done from rear end and deletion is done from front end. It is also known as **First-In-First-Out(FIFO)** data structure.

**Pictorial view of Queue**



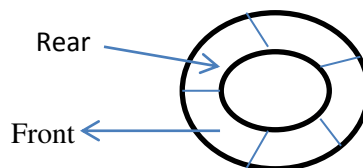
**Types of Queue:** There are three different types of queue based on how way they insert and delete from the queue.

1. **Linear Queue:** where insertion and deletion is done linearly i.e. insertion is done from one end and deletion is done from other end.

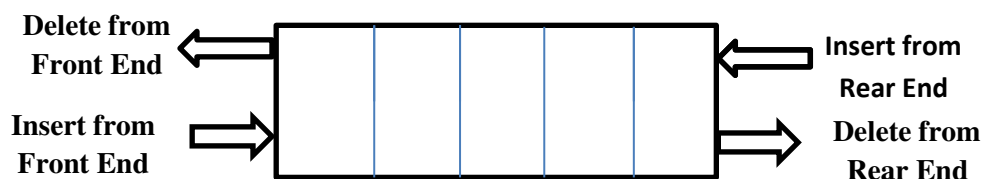
**Example:**



2. **Circular Queue:** where insertion and deletion of data items is done in circular fashion.



3. **Double Ended Queue:** Where insertion and deletion is done from both the ends i.e., insertion can be done from both rear end and front end, similarly deletion can also be done from both ends.



#### Applications of Queue:

- i. It is used in implementing job scheduling
- ii. It is used in implementing CPU scheduling
- iii. It is used in implementing Disk scheduling
- iv. It is used in implementing Input and Output buffer
- v. It is used in servicing printing requests of printer

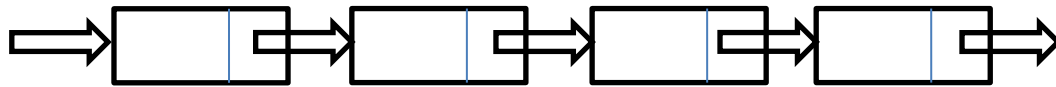
#### Various Operations of Queue:

1. **Insert\_Rear():** this function is used to insert a data item at rear end of the queue.
2. **Delete\_Front():** this function is used to delete a data item from front end of the queue.
3. **Display():** this function is used to list or print all the data items in the queue.
4. **QueueOverflow():** This function checks whether queue is full or not.
5. **QueueUnderflow():** This function checks whether queue is empty or not.

**c. Linked List:**

It is a linear data structure, where insertion and deletion are done in linear fashion. It is a collection of data items, where each data item is linked to another. It is a chain of nodes.

**Pictorial view of linked list:**



**Types of Linked List:**

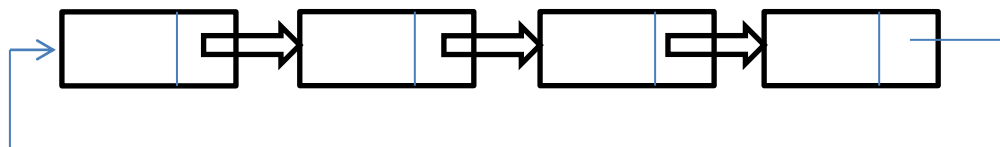
1. **Linear Linked List:** It is a linear singly linked list, where each node contains a pointer to the next node.

**Example:**



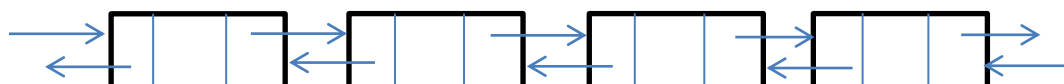
2. **Circular Linked list:** It is a linear data structure, where each node contains a pointer to the next node and last node contains a pointer to the first node.

**Example:**



3. **Doubly linked list:** It is a linear data structure, each node contains a pointer to the next node as well as to the previous node.

**Example**



**Applications of Linked List:**

1. It is used for implementing polynomials
2. It is used for implementing sparse matrix
3. It is used for implementing stack and queue

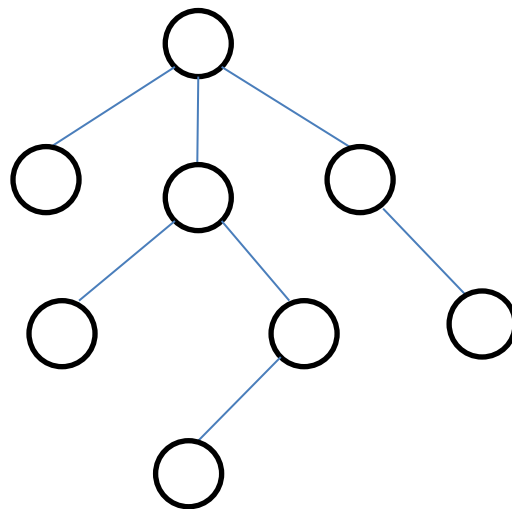
4. It is used for implementing SYMTAB and OPTAB in system software
5. It is used in implementing adjacency list representation of graphs.
6. It is used in implementing reverse of a number and string

**Various operations of linked list:**

1. **Insert():** this function can be used insert a data item at front of the list, at particular position of the list, or at end of the list.
2. **Delete():** this function can be used Delete a data item from front of the list, from a particular position of the list, or from end of the list.
3. **Search():** this function is used to search or find a particular data item in the list.
4. **Display():** this function is used to list or print all the data items in the list.

d. **Tree:** Tree is a non-linear data structure, where the data are arranged in hierarchical fashion.

**Pictorial View of Tree:**



**Types of Tree:**

1. Binary Tree
2. Binary Search Tree
3. RED-BLACK Tree
4. AVL Tree
5. B Tree
6. B+ Tree

**Applications of Linked List:**

1. It is used for implementing various data compression techniques
2. It is used for implementing routing table in networks
3. It is used for implementing graphs and maps

4. It is used for implementing social networks
5. It is used for implementing expression parser or tree
6. It is used for implementing evaluation of expressions
7. It is used in database such as indexing

**Various operations of linked list:**

1. **Insert():** This function is used insert a data item in the tree
2. **Delete():** This function is used Delete a data item from the tree
3. **Search():** This function is used to search or find a particular data item in the tree.
4. **Display():** This function is used to list or print all the data items in the tree.