# COURSE NOTES

FOR

## Bachelor Computer Applications

First Semester

## Programming Principles & Algorithms

as per syllabus of

**Mahatma Gandhi Kashi Vidyapith, Varanasi**

Prepared By:

**Department of Computer Science**
**Microtek College of Management & Technology**
**Varanasi.**

# BCA-S102T Programming Principles & Algorithms

**UNIT-I**
Introduction to 'C' Language
History, Structures of 'C' Programming, Function as building blocks.
Language Fundamentals
Character set, C Tokens, Keywords, Identifiers, Variables, Constant, Data Types, Comments.

**UNIT-II**
Operators
Types of operators, Precedence and Associativity, Expression, Statement and types of statements Built in Operators and functions
Console based I/O and related built in I/O function: printf(), scanf(), getch(), getchar(), putchar(); Concept of header files, Preprocessor directives: #include, #define.

**UNIT-III**
***Control structures***
Decision making structures: If, If-else, Nested If-else, Switch; Loop Control structures: While, Do-while, for, Nested for loop; Other statements: break, continue, goto, exit.

**UNIT-IV**
***Introduction to problem solving***
Concept: problem solving, Problem solving techniques (Trial & Error, Brain Storming, Divide & Conquer)
Steps in problem solving (Define Problem, Analyze Problem, Explore Solution) Algorithms and Flowcharts (Definitions, Symbols), Characteristics of an algorithm ,Conditionals in pseudo-code, Loops in pseudo code Time complexity: Big-Oh notation, efficiency Simple Examples: Algorithms and flowcharts (Real Life Examples)

**UNIT-V**
***Simple Arithmetic Problems*** Addition / Multiplication of integers, Determining if a number is +ve / -ve / even / odd, Maximum of 2 numbers, 3 numbers, Sum of first n numbers, given n numbers, Integer division, Digit reversing, Table generation for n, $a^b$, Factorial, sine series, cosine series,Pascal Triangle, Prime number, Factors of a number, Other problems such as Perfect number, GCD numbers etc (Write algorithms and draw flowchart), Swapping

**UNIT-VI**
***Functions***
Basic types of function, Declaration and definition, Function call, Types of function, Parameter passing, Call by value, Call by reference, Scope of variable, Storage classes, Recursion.

Reference Books :
1. Let us C-Yashwant Kanetkar.
2. Programming in C-Balguruswamy
3. C in Depth– S.K. Srivastava,Deepali Srivastava

# UNIT - I

## History of C Language

One such attempt was development of a language called Combined Programming Language (CPL) at Cambridge University in 1963. However, it turned out to be too complex, hard to learn, and difficult to implement. Subsequently, in 1967, a subset of CPL, Basic CPL (BCPL) was developed by Martin Richards incorporating only the essential features. However, it was also not found to be sufficiently powerful. Around the same time, in 1970, another subset of CPL, a language called B was developed by Ken Thompson at Bell Labs. However, it turned out to be not sufficient in general**. In 1972, Dennis Ritchie at Bell Labs developed C language** incorporating best features of both BCPL and B languages.

## Features of C Language

**C is often termed as a middle level programming language** because it combines the power of a high-level language with the flexibility of a low-level language. C is designed to have a good balance between both extremes. Programs written in C give relatively high machine efficiency as compared to high-level languages. Similarly, C language programs provide relatively high programming efficiency as compared to low-level languages.

## Why is C Language Popular

There are several features which make C, a suitable language to write system programs. These are:

- C is a machine independent and highly portable language.
- It. is easy to learn, it has only 32 keywords.
- Users can create their own functions and add to C library to perform a variety of tasks.
- C language allows manipulation of BITS, BYTES, and ADDRESSES.
- It has a large library of functions.

## Components of C Language (Tokens)

A Token is a smallest element in a program that is meaningful to the computer. These Tokens define the structure of the language.

The five main components (tokens) of 'C' language are:

- The character set
- The Data types
- Constants
- Variables
- Keywords

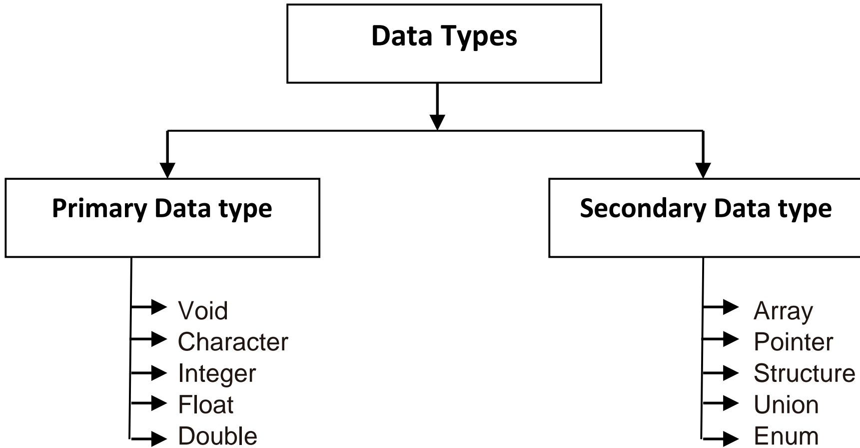
## The Character Set

Any alphabet, digits or special symbol used to represent information is denoted by character. The characters in C are grouped into four categories:

- Letters — A - - - Z or a - z
- Digits — 0,1,----9
- Special Symbols — -.’@#%'" &*() _-+ = I\{}[]:;"'< > , . ?/.
- White spaces — blank space, horizontal tab, carriage return, new line, and form feed.

## The Data Types

The power of a programming language depends, among other things, on the range of different types of data it can handle. Data values passed in a program may be of different types. The C data types can be broadly divided into two categories.




## Primary Data Types

There are five primary data types in C language.

- char — Char data type is used to store a single character belonging to the defined character set of 'C' language.
- int — int data type is used to store signed integers, for example, positive or negative integers.
- float — float data type is used to store real numbers with single. precision (precision of six digits after decimal points).
- double — double data type stores real numbers with double precision, that is, twice the storage space required by float.
- void — void data type is used to specify empty set containing no values.

- **Character**

| Data Type | Storage Space | Format | Range of Values |
|---|---|---|---|
| Char | 1 byte | %c | ASCII character set (-128 to 127) |
| unsigned char | 1 byte | %c | ASCII character set ( 0 to 255 ) |

- **Integer**

| Data Type | Storage Space | Format | Range of Values |
|---|---|---|---|
| Int | 2 bytes | %d ,%i | **-32768 to +32767** |
| unsigned int | 2 bytes | %u | **0 to 65535** |
| long int | 4 bytes | %ld | **-2147483648 to +2147483648** |
| long unsigned int | 4 bytes | %lu | **0 to 4,294,967,295** |

- **Real Numbers**

| Data Type | Storage Space | Format | Range of Values |
|---|---|---|---|
| Float | 4 bytes | %f | **-3.4*1038 to +3.4*1038** |
| Double | 8 bytes | %lf | **$-1.7*10^{308}$ to $+1.7*10^{308}$** |
| long double | 10 bytes | %Lf | **$-1.7*10^{4932}$ to $+1.7*10^{4932}$** |

## Variables

Variables are the data items whose values may vary during the execution of the program. A specific location or address in the memory is allocated for each variable and value of that variable is stored in that location.

## Rules for declaring variable name

- Variable name may be a combination of alphabet, digits, or underscores and its length should not exceed eight characters.
- First character must be an alphabet.
- No commas or blank spaces are allowed in variable name.
- Among the special symbols, only underscore can be used in variable name.
- Example: emp_age and item_ 4

## Variable Declaration and values Assignment

All the variables must be declared before their use. Declaration does two things:

- It tells the compiler what the variable name is.
- It specifies the type of data, the variable will hold.

A variable declaration has the form:

```
type_specifier list_of_variables;
```

here type_specifier is one of the valid data types. List_ of_variables is a comma separated list of identifiers representing the program variables.

**Examples:**

```
int a,b,c;
char ch;
```

To assign values to the variable, assignment operator (=) is used. Assignment is of the form

```
variable_name = value;
```

**Example:**

```
int  A,B;
A=10;
B=50;
```

It is also possible to assign a value to the variable at the time of declaration.

```
Data type variable_name=value;
```

**Example**

```
int  A=10;
char ch= '@';
```

## Constants

Constants are the fixed values that remain unchanged during the execution of a program and are used in assignment statements. Constants are stored in variables.

To declare any constant, the syntax is

```
const data type var_name = value ;
```

In 'C' language, there are five types of constants.

| | | |
|---|---|---|
| 1 | character | character constant consist of a single character, single digit, or a single special symbol enclosed within a pair of single inverted commas. i.e. 'A','%' |
| 2 | integer | An integer constant refers to a sequence of digits. There are. Three types of integers: decimal, octal and hexadecimal. In octal notation, write (0) immediately before the octal representation. For example: 0.76, -076.<br>In hexadecimal notation, the constant is preceded by 0x. Example: 0x3E, -0x3E. No commas or blanks are allowed in integer constants. |
| 3 | real | A real constants consist of three parts : Sign (+ or 0) , Number portion (base), exponent portion<br>i.e. +.72 , +72 , +7.6E+2 , 24.3e-5 |
| 4 | string | A string constant is a sequence of one or more characters enclosed within a pair of double quotes (" "). If a single character is enclosed within a pair of double quotes, it will also be interpreted as a string constant.<br>Examples:<br>"Welcome To Microtek \ n" , "a" , "123" |
| 5 | logical | A logical constant can have either a true value or a false value. In 'C all the non zero values are treated as true value while 0 is treated as false. |

## Keywords

Keywords are the words which have been assigned specific meaning in the context of C language programs. Keyword should not be used as variable names to avoid problems.

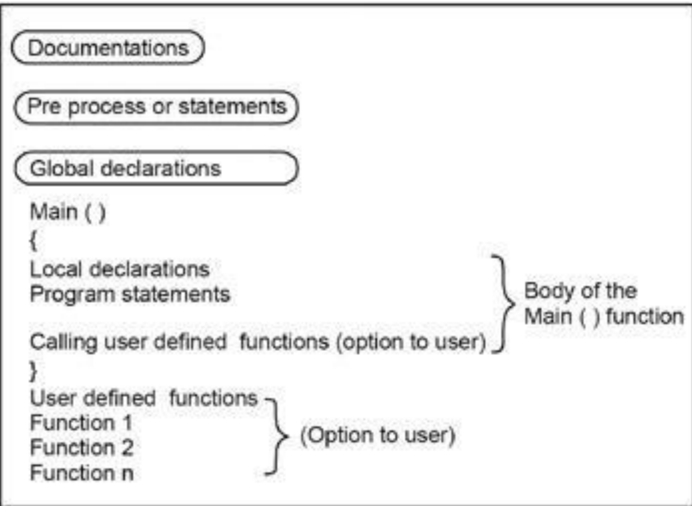
A keyword is a word or identifier that has a particular meaning to the programming language.

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

# Structure of a C program

Every C program consists of one or more functions. A function is nothing but a group or sequence of C statements that are executed together. Each C program function performs a specific task. The '**main()**' function is the most important function and **must** be present in every C program. The execution of a C program begins in the **main()** function.

The figure below shows the structure of a C program.




### Documentations

The documentation section consist of a set of comment lines giving the name of the program, the another name and other details, which the programmer would like to use later.

### Preprocessor Statements

The preprocessor statement begin with # symbol and are also called the preprocessor directive. These statements instruct the compiler to include C preprocessors such as header files and symbolic constants before compiling the C program.

### Global-Declarations

The variables are declared before the main ( ) function as well as user defined functions are called global variables. These global variables can be accessed by all the user defined functions including main ( ) function.

### The main ( ) function

Each and Every C program should contain only one main ( ). The C program execution starts with main ( ) function. No C program is executed without the main function. The main ( ) function should be written in small (lowercase) letters and it should not be terminated by semicolon. Main ( ) executes user defined program statements

### Braces

Every C program should have a pair of curly braces ({, }). The left braces indicates the beginning of the main ( ) function and the right braces indicates the end of the main ( )

function. These braces can also be used to indicate the user-defined functions beginning and ending. These two braces can also be used in compound statements.

### Local Declarations

The variable declaration is a part of C program and all the variables are used in main ( ) function should be declared in the local declaration section is called local variables. Not only variables, we can also declare arrays, functions, pointers etc. These variables can also be initialized with basic data types.

### Program statements

These statements are building blocks of a program. They represent instructions to the computer to perform a specific task (operations). it also includes comments that are enclosed within /* and */ . The comment statements are not compiled and executed and each executable statement should be terminated with semicolon.

### User defined functions

These are subprograms, generally, a subprogram is a function and these functions are written by the user are called “user defined functions”. These functions are used to perform user specific tasks and this also contains set of program statements. They may be written before or after a main () function and called within main () function. This is an optional to the programmer.
i.e.

```
main()                  function1()                 function2()

{                       {                           {
       statement1;             statement1;                  statement1;
       statement2;             statement2;                  statement2;
       ……;                    …….. ;                       ……;
}                       }                           }
```

## Writing your first C program

```
/*  Author: SABAB ALI KHAN
    Purpose: This program prints a message
*/
1. #include<stdio.h>
2. main()
3. {
4.      printf(“Hello, world !”);
5. }
```

Type this program in any text editor and then compile and run it using a C-compiler. However, your task will become much easier if you are using an IDE such as Turbo C.

## How to Run your Program.

1. Go to Start then select Run option
2. Open command prompt ( type cmd and press Enter Key / press OK )

3. Go to the directory where you have installed Turbo C using command window.
4. Type TC at the DOS command prompt.
5. In the edit window that opens, type the mentioned program above.
6. Save the program as hello.c by pressing F2 or Alt + 'S'.
7. Press Alt + 'C' or Alt + F9 to compile the program.
8. Press Alt + 'R' or Ctrl + F9 to execute the program.
9. Press Alt + F5 to see the output.

Programmers are free to name C program functions (except the main() function).

***Understanding the program***

In the program you saw above, the information enclosed between '/* */' is called a 'comment' and may appear anywhere in a C program. Comments are optional and are used to increase the readability of the program.

The '#include' in the first line of the program is called a *preprocessor* directive.

'stdio.h' refers to a file supplied along with the C compiler. It contains ordinary C statements. These statements give information about many other functions that perform input-output roles. Thus, the statement '#include<stdio.h>' effectively inserts the file 'stdio.h' into the program file.

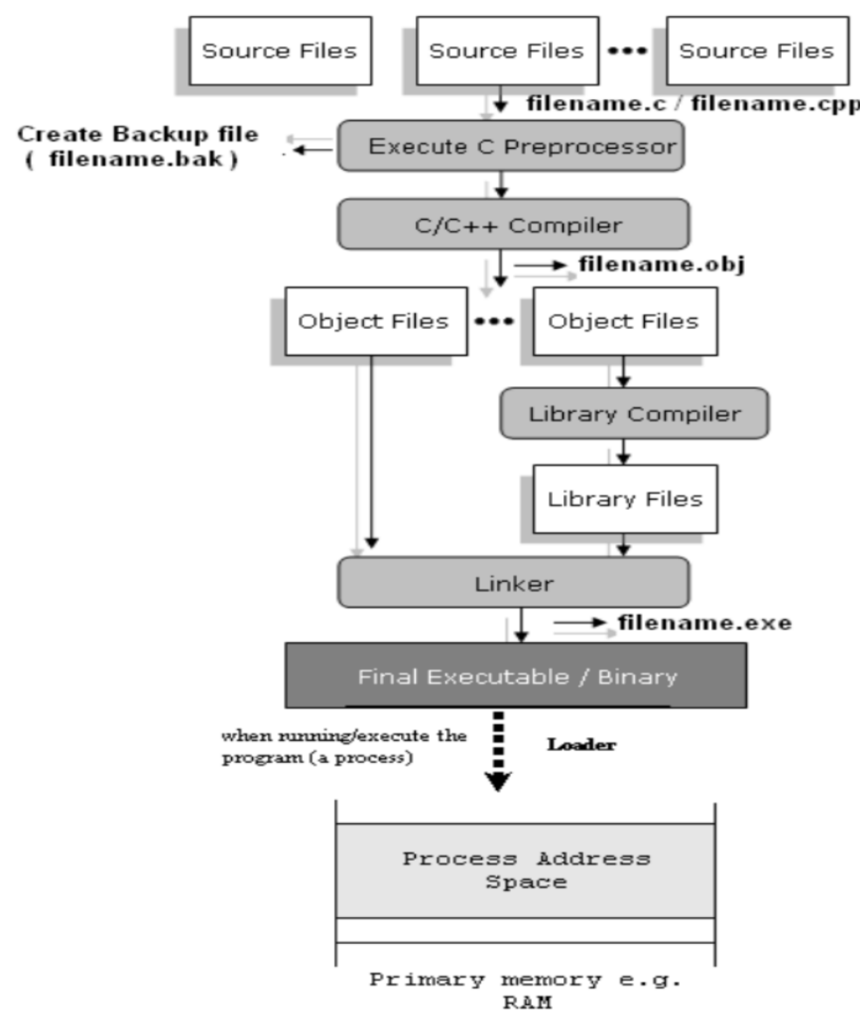
The next statement is the **main()** function. As you already know, this is the place where the execution of the C program begins.

Next comes the opening brace '{', which indicates the beginning of the function. The closing brace '}' indicates the end of the function.

The statement **printf()** enclosed within the braces'{}' informs the compiler to print (on the screen) the message enclosed between the pair of double quotes. In this case,

'**Hello, world !**' is printed.

# Compilation Process of 'C' Programs



# UNIT-II

## Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation on data stored in variables. The variables that are operated are termed as operand.

C operators can, be classified into a number of categories. They include:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Special operators

### Arithmetic operators

C provides all the basic arithmetic operators. There are five arithmetic operators in C.

| Operator | Purpose |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Remainder after integer division |

The division operator (/) requires the second operand as non zero, though the operands not be integers.
The operator (%) is known as modulus operator. It produces the remainder after the division of the two operands. The second operands must be non zero.

Example: if a = 25, b = 4

then a + b = 29

a - b = 21

a * b = 100

a/b = 6 (decimal parts truncated)

a%b = 1

## Relational Operators

Relational operator is used to compare two operands to see whether they are equal to each other, unequal, or one is greater or lesser than the other.

The operands can be variable, constants, or expression and the result is a numerical value. There are six relational operators.

| Operator | Meaning |
|---|---|
| **==** | equality |
| **!=** | Not equal to |
| **<** | less than |
| **>** | greater than |
| **< =** | less than or equal to |
| **> =** | greater than or equal to |

A simple relation contains only one relational operator and takes the following form:

```
ae-1 relational operator ae-2
```

ae-1 and ae-2 are **arithmetic expressions**, which may be simple constants, variables, or combination of these. The value of relational operator is either 1 or O. If the relation is - true, result is 1 otherwise it is O.

Example:

| Expressions | Result |
|---|---|
| 4.5 <= 10 | 1 (True) |
| 4.5 < -10 | 0 (False) |
| -35 >= 0 | 0 (False) |
| 10 < 7+5 | 1 (True) |

## Logical Operators

Logical operators are used to combine two or more relational expressions. C provides three different logical operators.

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>&&</td><td>Logical and</td></tr>
<tr><td>||</td><td>Logical or</td></tr>
<tr><td>!</td><td>Logical not</td></tr>
</table>

## Logical AND ( &&)

The result of Logical AND will be true only if both operands are true.

| Result_exp1 | Result_exp2 | Final Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

e.g. if ( 10 > 15 && 5 < 10) → Result will be 0 (false)

if ( 5 < 10 && 10 > 6 ) → Result will be 1 (true)

Explanation:

10 > 15 && 5 < 10

**0** && 1

0 ( false )

## Logical OR ( || )

The result of Logical OR will be true if any one operand is true

| Result_exp1 | Result_exp2 | Final Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

e.g. if ( 10 > 15 || 5 < 10) → Result will be 1 (True)

if ( 5 > 10 || 9 < 6 ) → Result will be 0 (false)

if ( 5 > 10 || 2 > 6 ) → Result will be 0 (false)

Explanation: 10 > 15 || 5 < 10

**0** || 1

1 (true)

**Logical NOT ( ! )**

Logical NOT ( ! ) is used to reverse the value of the expression.

| Expr_Result | Final Result |
|---|---|
| 0 | 1 |
| 1 | 0 |

e.g. if ( ! ( 5 > 10) ) → Result will be 1 (TRUE)

if ( ! ( 10 > 15 && 5 < 10) ) → Result will be 1 (TRUE)

if ( ! ( 5 > 10 || 9 < 6 ) ) → Result will be 1 (TRUE)

Explanation:

```
!  ( 5 > 10)
    \_____/
 !     0    (flase)
 \______/
   1 (true)
```

## Assignment Operator

Assignment operators are used to assign the result of an expression to a variable. The most commonly used assignment operator is (=).

An expression with assignment operator is of the following form:

Identifier = expression;

Example:

```
#include <stdio.h>

void main()
{
    int i ;
    i = 5;
    printf ("%d", i);
    i = i + 10;
    printf ("\n%d", i):
}
```

Output will be : 5
10

Expressions like i = i+10; , i = i-5; , i = i*2; , i = i/6, and i = i% 10 can be rewritten using shorthand assignment operators.

The shorthand assignment operators are of following type:

| V *op* = expression; |
|---|

This is equivalent to

```
V = V op expression;
```

Example: I = i+5; is equivalent to **i +** = 5;

I = i * ( y + 5 ); is equivalent to **i** * = ( y + 5 );

## Increment and Decrement Operators

'C' has two very useful operators + + and -- called increment and decrement operators respectively. These are generally not found in other languages. These operators are called unary operators as they require only one operand. This operand should necessarily be variables not constant.

The increment operator (+ +) adds one to the operand while the decrement operator (--) subtracts one from the operand.

These operators may be used in two ways.

1) **Prefix :**

When the operator used before the operand, it is termed as prefix.

e.g. ++A , --B

in this case the value of operand follow First Change Then Use ( F.C.T.U) concept

2) **Postfix:**

When the operator used before the operand, it is termed as prefix.

e.g. A++ , B--

in this case the value of operand follow First Use Then Change ( F.U.T.C) concept

Example:

**Postfix**

```
int N=10, R;
R = N++;  // post increment
printf("R=%d \n N=%d ", R , N);
```

it will produce **output** :

R=10 (Because before increment, value will assign first)
N=11

**Postfix**

```
int N=10, R;
R = ++N;  // pre increment
printf("R=%d \n N=%d ", R , N);
```

it will produce **output:**

R=11 (Because value will increment first, then value will assign)
N=11

Example 1:

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

```
void main()
{
   int  R, N=10;
   clrscr();
   R =  ++N  +  --N  +  --N  + N++   +  --N ;
  printf( " R= %d \n N=%d" , R,N);
  getch();
}
```

The **output** will be :

```
R=40
N=9
```

**Explanation:**

In this example there are for than one increment or decrement expressions are used, so that it follows the execution order (**prefix → operation → postfix).**

in this statement ( ++N + --N + --N + N++ + --N ; ) all prefix expression execute first ,then it perform operations like addition/subtraction or assignment etc. and after that it perform all postfix operations. In the operation the value of all operand will give last modified value by prefix operations.

## Conditional or Ternary Operator

A ternary operator is one which contains three operands. The only ternary operator available in C language is conditional operator pair " **? :** " . It is of the form

exp1 ? exp2 : exp3 ;

This operator works as follows. Exp1 is evaluated first. If the result is true then exp2 is executed otherwise exp3 is executed.

**Example 1:**

```
a = 10;
b = 15;
x = (a > b ? a : b);
```

in this expression value of b will be assigned to x.

Example 2:

```
a = 10;
b = 15;
x = (a > b )? printf ("First value is Greater ") : printf ("Second value is greater");
```

in this expression the result will be-

**Second value is greater**

## Bitwise operators

Bitwise operators are used to manipulate of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double data type.

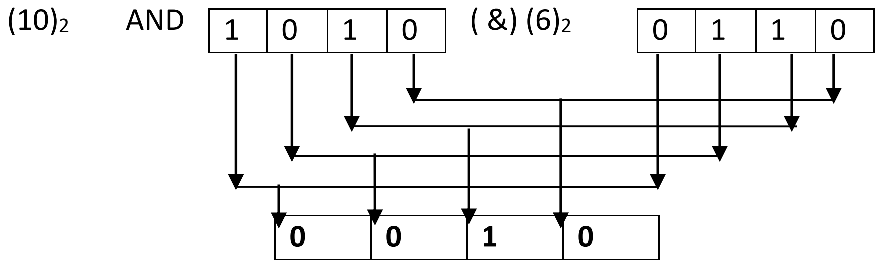
| **Some Bitwise Operators** |
|---|

| Operator | Meaning |
|---|---|
| & | Bitwise Logical AND |
| \| | Bitwise Logical OR |
| ^ | Bitwise Logical XOR |
| << | Left shift |
| >> | Right shift |
| ~ | One's complement |

**Bitwise Logical AND ( & )**

e.g.

```
int  N=10;
N= N & 6 ;
Printf("N=%d",N);
```



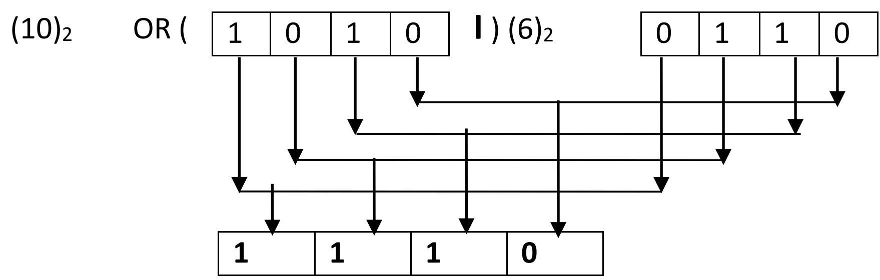

This will produce output:

N=2

**Bitwise Logical OR ( | )**

e.g.

```
int  N=10;
N= N | 6 ;
Printf("N=%d",N);
```



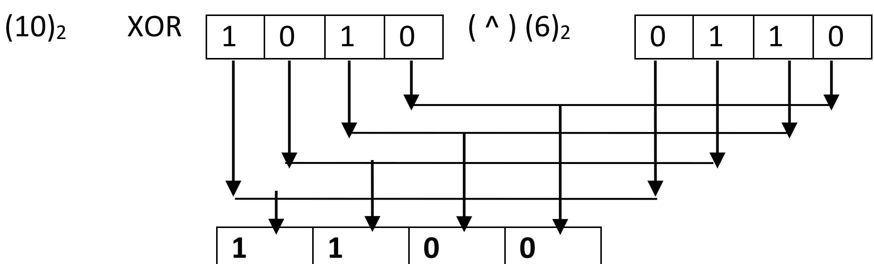
This will produce output: N=14

**Bitwise XOR ( ^ )**

The Bitwise XOR operator produce result 1 if both bits are different otherwise it will produce 0.

e.g.

```
int  N=10;
N= N ^ 6 ;
Printf("N=%d",N);
```




This will produce output:

N=12

**Shift Operators**

Shift operators use a shift distance ( number of shifts ) as the right-hand operand and the value which is to be shifted as the left-hand operand.

**Left Shift ( << )**

Left shift operator shifted bits left. When bits are shifted left, zero is filled in from right.
Each left-shift corresponds to multiplication of the value by 2.

It will give result $\mathbf{V=V*2^n}$ , where **n** is the number of bits to be shifted and **v** is the value to be shifted.

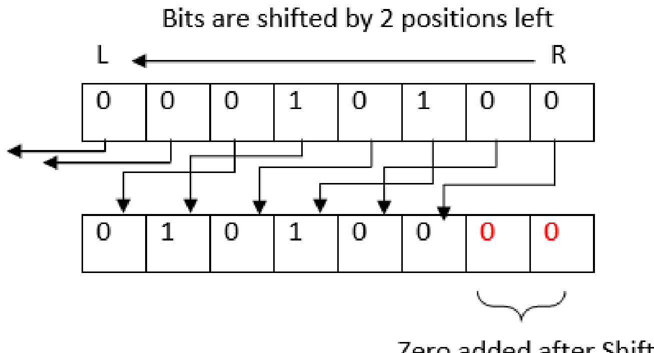
e.g.

```
int N=20;
N=N<<2;
```

it will produce Value

$N=N*2^2$
$N=20 * 2^2$
$N=20 * 4$
$N=80$

$(20)_{10}$ ➔ $(1\,0\,1\,0\,0)_2$




After Shifting the Value $(20)_{10}$ ➔ ( 0 0 0 1 0 1 0 0 $)_2$ will be $(80)_{10}$ ➔ ( 0 1 0 1 0 0 0 0 $)_2$

**Right Shift ( >> )**

Right shift operator shifted bits right . when bits are shifted right, sign bit are filled in from left. Each right-shift corresponds to division of the value by 2.

It will give result **$V=V/2^n$** , where **n** is the number of bits to be shifted and **v** is the value to be shifted.

e.g.

int N=20; it will produce Value

N=N>>2;

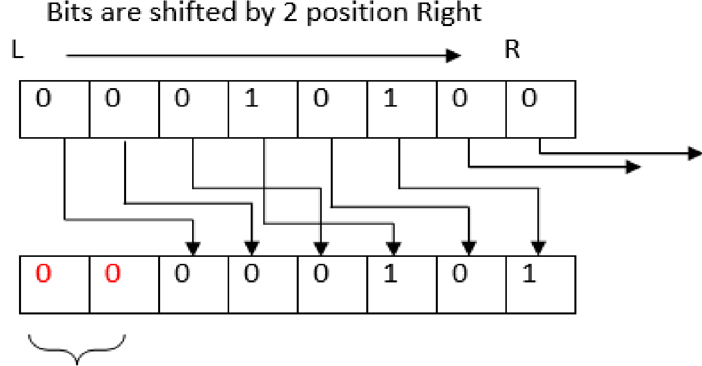
$N=N/2^2$

$N=20 / 2^2$

N=20 / 4

N=5

$(20)_{10}$ ➔ (1 0 1 0 0$)_2$




**Sign bit** ( 0 ) added after Shifting

Sign bit for Positive number is zero ( 0 ) , and for Negative number ( 1 )

After Shifting the Value $(20)_{10}$ ➔ ( 0 0 0 1 0 1 0 0 $)_2$ will be $(5)_{10}$ ➔ ( 0 0 0 0 0 1 0 1 $)_2$

## One's Complement Operator: ~

The one's complement operator (~), sometimes called the "bitwise complement" operator, yields a bitwise one's complement of its operand. That is, every bit that is 1 in the operand is 0 in the result. Conversely, every bit that is 0 in the operand is 1 in the result. The operand to the one's complement operator must be an integral type.

**Example**

let us take decimal no. **10**.First this number is converted to binary equivalent i.e. **0000000000001010**,taking one's complement, the number becomes **1111111111110101**.

**Example**

| Decimal | Original number(j) | One's Complement(m) |
|---|---|---|
| 1 | 0000000000000001 | 1111111111111110 |
| 2 | 0000000000000010 | 1111111111111101 |
| 3 | 0000000000000100 | 1111111111111011 |
| 4 | 0000000000001000 | 1111111111110111 |

# Special operators

'C' language supports some special operators such as comma operator, sizeof operator, pointer operators (& and *), and member selection operators (. and ->). Pointer operators will be discussed while introducing pointers while member selection operator will be discussed with structures and union. Let us discuss comma operator and sizeof operator.

## comma operator

This operator is used to link the related expressions together.

Example:

intval, x, y;

value = (x= 10, y = 5, x+y);

it first assigns 10 to x then 5 to y finally sum x + y to value.

## sizeof operator

The sizeof operator is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, constant, or a data type qualifier.

Example:

int n;

```
n = sizeof (int);
printf ("n=%d \n", n);
n = sizeof (double);
printf ("n=%d", n);
```

Output: n = 2
n = 8

# Operator precedence

Precedence defines the sequence in which operators are to be applied on the operands while evaluating the expressions involving more than one operator. Operators of same precedence are evaluated from left to right or right to left, depending upon the level. This is known as associativity property of an operator.

| DESCRIPTION | OPERATORS | ASSOCIATIVITY |
|---|---|---|
| Function expression | ( ) | L → R |
| Array expression | [ ] | L → R |
| Structure operator | -> | L → R |
| Structure operator | . | L → R |
| Unary Minus | – | R → L |
| Increment / Decrement | ++ -- | R → L |
| One's complement | ~ | R → L |
| Negation | ! | R → L |
| Address of | & | R → L |
| value at address | * | R → L |
| Type cast | (type) | R → L |
| Size in bytes | sizeof | R → L |
| Multiplication | * | L → R |
| Division | / | L → R |
| Modulus | % | L → R |
| Addition | + | L → R |
| Subtraction | – | L → R |
| Left shift | << | L → R |
| Right shift | >> | L → R |
| Less than | < | L → R |
| Less than or equal to | <= | L → R |
| Greater than | > | L → R |
| Greater than or equal to | >= | L → R |

| | | |
|---|---|---|
| equal to<br>Not equal to | ==<br>!= | L ➔ R<br>L ➔ R |
| Bitwise AND<br>Bitwise XOR | &<br>^ | L ➔ R<br>L ➔ R |
| Bitwise OR | \| | L ➔ R |
| Logical AND<br>Logical OR | &&<br>\|\| | L ➔ R<br>L ➔ R |
| Conditional | ?: | R ➔ L |
| Assignment | =<br>*= /= %=<br>+= -= &=<br>^ = \|=<br><< = >> = | R ➔ L<br>R ➔ L<br>R ➔ L<br>R ➔ L<br>R ➔ L |
| Comma | , | R ➔ L |

# Expressions

An expression is a combination of variables, constants, and operators arranged according to syntax of the language. Some examples of expressions are:

e.g.
c = (m + n) * (a - b);
temp = (a + b + 'c) / (d - c);

Expression is evaluated by using assignment statement.
Such a statement is of the form

| Variable = expression ; |
|---|

The expression is evaluated first, then the value is assigned to the variable left hand side.
But all the relevant variables must be assigned the values before evaluation of the expression.

# Type conversion in Expressions

## Automatic Type conversion :

If the operands are different types, the lower type is automatically converted to the higher type before the operation proceeds. The result is of the higher type.
Given below is the sequence of rules that are applied while evaluating expressions.

| Op-I | Op-2 | Result |
|---|---|---|
| long double | any | long double |
| double | any | double |
| float | any | float |
| unsigned long int | any | unsigned long int |
| long int | any | long int |
| unsigned int | any | unsigned int |

The final result of an expression is converted to the type of the variable on the left of the assignment sign before 'assigning value to it.

However, the following changes are introduced during the final assignment.

- float to int causes truncation of the fractional part.
- double to float causes rounding of digits.
- long int to int causes dropping of the excess higher order bits .

## Casting a value

Casting a value is forcing a type conversion in a way that is different from the auto conversion. The process is called type cast. The general form of casting is

(type_desired) expression;

where type_desired: standard C data types and expression : constant, variable or expression.

**Example:**

```
#include <stdio.h>
void main()
{
        int total_marks=500,ob_marks=234;
        float per1 , per2;
        per1 = (ob_marks /  total_marks) * 100;
        per2 = ( float ) (ob_marks / total_marks) * 100;
        printf(" Percentage without type casting = %.2f",per1);
        printf(" Percentage  After    type casting = %.2f",per2);
        getch();
}
```

The Output will be:

```
Percentage without type casting = 0.00
Percentage  After    type casting = 46.80
```

in expression per2 = **( float )** (ob_marks / total_marks) * 100; division is converted to float, otherwise decimal part of the result of division would be lost and per1 would represent a wrong figure or zero.

# Introduction to Input/Output

Input refers to accepting of data while output refers to the presentation of data. Normally the data is accepted from keyboard and is outputted on to the screen.

C language has a series of standard input/output (I/O) functions. Such I/O functions together form a library named stdio.h. Irrespective of the version of 'C' language, user will have access to all such library functions. These library functions are classified into three broad categories.

a) Console I/O functions — functions which accept input from keyboard and produce output on the screen.

b) Disk I/O functions — functions which perform I/O operations on secondary storage devices like floppy disks or hard disks.

c) Port I/O functions — functions which perform I/O operations on various ports like printer port, and mouse port.

## Console I/O Functions

Console I/O refers to the operations that occur on the keyboard and the screen of your computer. Console I/O functions can be further classified as

- **Formatted Console, Input / Output**
- **Unformatted Console Input / Output**

| Console Input/Output functions | | | | | |
|---|---|---|---|---|---|
| **Formatted functions** | | | **Unformatted functions** | | |
| **Type** | **put** | **utput** | **/pe** | **put** | **utput** |
| char | scanf( ) | printf( ) | char | getch( )<br>getche( )<br>getchar( ) | putch( )<br>putchar( ) |
| int | scanf( ) | printf( ) | int | - | - |
| float | scanf( ) | printf( ) | float | - | - |
| string | scanf( ) | printf( ) | string | gets( ) | puts( ) |

### Formatted Console I/O Functions

As can be seen from Figure 11.1 the functions **printf( ),** and **scanf( )** fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form.

- **Formatted Output**

The **printf( )** statement provides certain features through which the screen output is effectively controlled. The general form of printf( ) function is:
printf ( "format string", list of variables ) ;

The format string can contain:

a) Characters that are simply printed as they are
b) Conversion specifications that begin with a % sign
c) Escape sequences that begin with a \ sign

**For example**, look at the following program:

```
main( )
{
int avg = 346 ;
float per = 69.2 ;
printf ( "Average = %d\nPercentage = %f", avg, per ) ;
}
```

The output of the program would be...

```
Average = 346
Percentage = 69.200000
```

## Conversion Specifications

The conversion specifications are used to provide the type and size of the data. Each conversion specification must begin with %.

In the above example %d and %f are the conversion characters.

The general form of conversion specifier is

**% fws fx**

Where fws = field width specifier
fx = format specifier

The field width specifier tells printf( ) how many columns on the screen should be used while printing a value.

**Example:** %10d tells to print the variable as a decimal integer in the field of 10 columns. If we include a minus sign in conversion specification (% - 10d), this means left justification is desired and the value will _ be padded with blanks on the right.

Given below is a list of conversion characters that can be used with the printf( ) function.

| Data Type | | Conversion Character |
|---|---|---|
| Integer | short signed | %d or % i |
| | short unsigned | %u |
| | Long signed | %ld |
| | Long unsigned | %lu |
| | unsigned Hexadecimal | %x |
| | unsigned octal | %0 |
| Real | float | %f |
| | double | % lf |
| character | signed character | %c |
| | unsigned character | %c |
| String | | %s |

## Escape Sequences

The backslash symbol (\) is considered as an escape character because it causes an ,escape from the normal interpretation of a string, so that the next character. is recognized as the one that has special meaning.

| Escape sequence | Purpose | Escape sequence | Purpose |
|---|---|---|---|
| \n | New line | \t | Tab |
| \b | Backspace | \r | Carriage return |
| \f | form feed | \a | Alert |
| \' | single quote | \" | double quote |
| | | \\ | backslash |

### Output of Integer Numbers

The format specification for printing an integer number is %wd where ' w' specifies minimum width for the output. The number is written right justified in the given field width.

**Example:**

printf ("%d", 12345);

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|

printf ("%10d", 12345);

| | | | | | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|

printf ("% 010d", 12345);

| 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|

printf ("% -10d", 12345);

| 1 | 2 | 3 | 4 | 5 | | | | | |
|---|---|---|---|---|---|---|---|---|---|

The Empty field will be fill with space.

### Output of Real Numbers

The real number is displayed in decimal notation with format specification **"% w.p f "** where 'w' , is the integer which represents the minimum number of positions that are to be used and 'p' indicates that In the total width, how many numbers will be placed after the decimal.

**Example:**

printf ("%7.4f " , 96.7654);

| **9** | **6** | **.** | **7** | **6** | **5** | **4** |
|---|---|---|---|---|---|---|

printf ("%7.2", 96.7654);

| | | **9** | **6** | **.** | **7** | **7** |
|---|---|---|---|---|---|---|

print ("%f', ,96.7654);

| **9** | **6** | **.** | **7** | **6** | **5** | **4** |
|---|---|---|---|---|---|---|

### Printing of String

The format specification for outputting strings is similar to that of real numbers. It is of the form '%w.p s ' where 'w' specifies the field width for display and 'p' specifies the number of characters to be displayed. The display is right justified.

Example: Let the string to be printed is "NEW DELHI 110001"

The total length is 16 characters (including blanks).

| specification | **output** | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| %s | N | E | W | | D | E | L | H | I | | 1 | 1 | 0 | 0 | 0 | 1 | | | | |
| % 20s | | | | | N | E | W | | D | E | L | H | I | | 1 | 1 | 0 | 0 | 0 | 1 |
| % 20.7s | | | | | | | | | | | | | | N | E | W | | D | E | L |

- ## Formatted Input

Formatted input refers to an input data that has been arranged in a particular format For the formatted input we use the function scanf( ).

**scanf( ) function**

scanf( ) function, allows us to read formatted data and automatically convert numeric

Information "into integers and float. The general from of scanf( ) is

**scanf ("control string", arg1, arg2, ..... );**

Control string specifies the field format in which data is to be entered and the arguments arg1, arg2 - - - - - specify the ADDRESS OF LOCATION where value is to be stored. Control string and arguments are separated by commas.

Given below is a list of format specifier used to read the inputs:

| Code | Meaning | Code | Meaning |
|---|---|---|---|
| %c | Read a single character .. | %d | Read a decimal integer |
| %ld | Read a long integer | %i | Read a decimal integer |
| %e | Read a floating point number | %f | Read a floating point number |
| %h | Read a short ·integer | %0 | Read an octal |
| %s | Read a string | %x | Read a hexadecimal number |
| %p | Read a pointer | %n | Reads an integer value equal to the *no.* of character read so far. |

## Input of Integer Numbers

The format specification for reading an integer number is % wd where (%) sign indicates conversion specification, w is the integer number for field width specification and d indicates that the number is to be read in integer mode.

Example: scanf ("% 2d % 5d", & n1, &n2);

An input field may be skipped by specifying & in place of field width.

Example: scanf ("% 2d % * d % 6d", &n1, &n2);

## Input of Character Strings

% ws or % wc can be used as the specification for reading character strings. The specifier %s terminates reading a string at the encounter of blank space. Some versions of scanf( ) supports the following conversion specification for strings.

**% [characters]   and   % [ ^ characters]**

The specification **% [characters]** means that only the characters specified within brackets are permissible in the input string. If the input string contains any other character, the string **will** be terminated at the first encounter of such a character.

The specification **% [ ^character]** does exactly the reverse, that is, character specified after circumflex (^) are not permitted.

Example :

**scanf(“ %1[ YN ]c ”, &ch);**

Input for ch Only permitted ‘Y’ or ‘N’ single character.

Example :

**scanf(“ %1[ ^YN ]c ”, &ch);**

Input for ch permitted all characters except ‘Y’ or ‘N’ single character.

## Unformatted Console I /O - Function

Unformatted console I/O functions cannot control the format of reading and writing the data. All the unformatted console -I/O functions are defined in stdio.h header file.

### Character Input Function

**1). getchar( ) function :-**

This is a un-formatted console Input function which is used to enter or Input One character at a time from a standard Input device eg. Keyboard . In this the entered character is echoed (display) on the screen and the use need to press Enter key.

eg. char alphabet ;
alphabet = getchar() ;

2). **getche() function :-**

This also comes in the unformatted console Input function which is used to enter ( input ) One character at a time . In this , the entered character is echoed / displayed on the screen but the user not need to press Enter key to submit the character . As the key is press, it is accepted from the program.

eg. char alphabet ;
alphabet = getche() ;

3). **getch() function :-**

This is also unformatted console Input function which is used to enter one character at a time . In this, the user need neither to press Enter key nor the character is echoed on screen . It is used where the user do not want to show the Input.

eg. char alphabet ;
alphabet = getch() ;

The main Difference between these functions are:

| Function | Wait for Enter Key? | Display Character during input, on Screen? |
|---|---|---|
| **getch()** | No | No |
| **getche()** | No | Yes |
| **getchar()** | Yes | Yes |

Example:
```
# include <conio.h>
# include <stdio.h>
void main( )
{
        char c;
        c = getchar( );  // or c=getche(); or c=getch();
        if (c == 'y' II c = = 'Y' )
        printf ("C is a character \n");
}
```

## Character Output Function

### putchar () :-

putchar is a macro defined as putc(c,stdout) putchar puts the character given by c on the output stream stdout(console). It is use to print one character (at a time) on the screen at the current cursor location.

syntax: putchar (variable_name);

Variable must be of character type.

eg.

```
main()
{
  char ch ;
  printf(" Enter a key ") ;
  ch = getch() ;
  printf(" \n You have enter character ") ;
  putchar( ch ) ;
}
```

### putch() function :-

putch outputs the character c to the current text window. It is a text-mode function that performs direct video output to the console.

eg.

```
main()
{
  char ch ;
  printf(" Enter a key ") ;
  ch = getch() ;
  printf(" \n You have enter character ") ;
```

```
  putch( ch ) ;
}
```

### Character String Input Function

The **gets( )** function is used to read a character entered on the keyboard and places them at the address pointed to by its character pointer argument. Characters are entered until the enter key is pressed.

Syntax: char * gets (char *a);

where “a” is the character array or character pointer.

### Character String Output Functions

The **puts( )** function writes its string argument to the screen followed by the newline.

Syntax: char * puts (const char * a);

puts( ) function takes less space then printf( ). It is faster than printf( ). It does not output numbers or do format conversions as puts( ) outputs character string only.

**Example:**

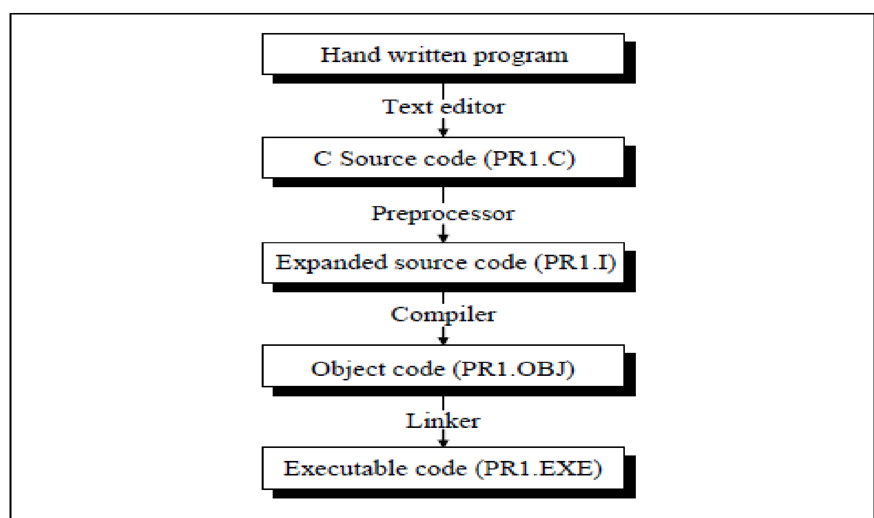
```
main()
{
 char name[ 15 ] ;
 clrscr() ;
 puts(" Enter name of 15 Characters: ") ;
 gets( name ) ;                          /*  String Input  */
 puts(" The name Entered is: ") ;
 puts(" name ") ;                        /*  String Output  */
 getch() ;
}
Output .
Enter name of 15 Characters: SABAB KHAN
 The name Entered is: SABAB KHAN
```

# Preprocessor

The C preprocessor is a program that processes our source program before it is passed to the compiler. Preprocessor commands (often known as directives) form what can almost be considered a language within C language. We can certainly write C programs without knowing anything about the preprocessor or its facilities.

# Features of C Preprocessor

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a '#' symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition.




| Processor | Input | Output |
|---|---|---|
| Editor | Program typed from keyboard | C source code containing program and preprocessor commands |
| Prepro-cessor | C source code file | Source code file with the preprocessing commands properly sorted out |
| Compiler | Source code file with preprocessing commands sorted out | Relocatable object code |
| Linker | Relocatable object code and the standard C library functions | Executable code in machine language |

We would learn the following preprocessor directives here:

(a) Macro expansion
(b) File inclusion
(c) Conditional Compilation
(d) Miscellaneous directives

# Macro Expansion (#define)

Have a look at the following program.

```
#define UPPER 25
main( )
{
  int  i ;
  for ( i = 1 ; i <= UPPER ; i++ )
  printf ( "\n%d", i ) ;
}
```

In this program instead of writing 25 in the for loop we are writing it in the form of UPPER, which has already been defined before main( ) through the statement,

```
#define UPPER 25
```

This statement is called 'macro definition' or more commonly, just a 'macro'. During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25. Here is another example of macro definition.

```
#define PI   3.1415
main( )
{
float  r = 6.25 ;
float  area ;

area = PI * r * r ;
printf ( "\nArea of circle = %f", area ) ;
}
```

UPPER and PI in the above programs are often called 'macro templates', whereas, 25 and 3.1415 are called their corresponding 'macro expansions'.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the #define directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

**Note** that a macro template and its macro expansion are separated by blanks or tabs. A space between # and define is optional. Remember that a macro definition is never to be terminated by a semicolon.

**Example :**

```
#define AND &&
#define OR   ||
main( )
{
```

```
int  f = 1, x = 4, y = 90 ;

if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
printf ( "\nYour PC will always work fine..." ) ;
else
printf ( "\nIn front of the maintenance man" ) ;
}
```

A **#define** directive could be used to replace even an entire C statement. This is shown below.

```
#define FOUND printf ( "The Yankee Doodle Virus" )
; main( )
{
  char  signature ;

 if ( signature == 'Y' )
 FOUND
 else
 printf ( "Safe... as yet !" ) ;
}
```

## Macros with Arguments

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

**Example:**

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
float  r1 = 6.25, r2 = 2.5, a ;
a = AREA ( r1 ) ;
printf ( "\nArea of circle = %f", a ) ;
a = AREA ( r2 ) ;
printf ( "\nArea of circle = %f", a ) ;
}
```

Here’s the output of the program...

Area of circle = 122.656250
Area of circle = 19.625000

After the above source code has passed through the preprocessor, what the compiler gets to work on will be this:

```
main( )
```

```
{
float  r1 = 6.25, r2 = 2.5, a ;

a = 3.14 * r1 *r1 ;
printf ( "Area of circle = %f\n", a ) ; a = 3.14 *r2 *
r2 ;
printf ( "Area of circle = %f", a ) ;
}
```

Here is another example of macros with arguments:

**Example 1:**

```
#define ISDIGIT(y) ( y >= 48 && y <= 57 )
main( )
{
char  ch ;

printf ( "Enter any digit " ) ;
scanf ( "%c", &ch ) ;

if ( ISDIGIT ( ch ) )
printf ( "\nYou entered a digit" ) ;
else
printf ( "\nIllegal input" ) ;
}
```

**Example 2:**

```
#define SQUARE(n) n * n
main( )
{
int  j ;

j= 64 / SQUARE ( 4 ) ;
printf ( "j = %d", j ) ;
}
```

The output of the above program would be:

j= 64

## *#undef* Directive

On some occasions it may be desirable to cause a defined name to become ‘undefined’. This can be accomplished by means of the #undef directive. In order to undefine a macro that has been earlier #defined, the directive,

#undef macro template

can be used. Thus the statement,

#undef PENTIUM

would cause the definition of PENTIUM to be removed from the system. All subsequent #ifdef PENTIUM statements would
evaluate to false. In practice seldom are you required to undefine a macro, but for some reason if you are required to, then you know that there is something to fall back upon.

## Macros versus Functions

In a macro call the preprocessor replaces the macro template with its macro expansion, in a stupid, unthinking, literal way. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function
is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program.

But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program.
This gets avoided with macros since they have already been expanded and placed in the source code before compilation.

# File Inclusion (#include)

The second preprocessor directive is file inclusion. This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

#include "filename"

and it simply causes the entire contents of filename to be inserted into the source code at that point in the program. Of course this presumes that the file being included is existing. It can be used in two cases:

(a) If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are

#include at the beginning of main program file.

(b) There are some functions and some macro definitions that we need almost in all

programs that we write. These commonly needed functions and macro definitions can be stored in a file, and that file can be included in every program we write, which would add all the statements in this file to our program as if we have typed them in.

It is common for the files that are to be included to have a **( .h** ) extension. This extension stands for 'header file', possibly because it contains statements which when included go to the head of your program.

Actually there exist two ways to write #include statement. These are:

```
#include "filename"
#include <filename>
```

The meaning of each of these forms is given below:

**#include "goto.c"**
This command would look for the file goto.c in the current directory as well as the specified list of directories as mentioned in the include search path that might have been setup.

**#include <goto.c>**

This command would look for the file goto.c in the specified list of directories (Search path) only.

If you are using Turbo C/C++ compiler then the search path can be setup by selecting 'Directories' from the 'Options' menu. On doing this a dialog box appears. In this dialog box against 'Include Directories' we can specify the search path. We can also specify multiple include paths separated by ';' (semicolon) as shown below:

c:\tc\lib ; c:\mylib ; d:\libfiles

The path can contain maximum of 127 characters. Both relative and absolute paths are valid. For example '..\dir\incfiles' is a valid path.

**Program :**
First create two programs

**prog1.h** (header file)

```
#include<stdio.h>

void  display()
{
printf("\nThis is prog1 header  file\n");
printf("Welcome to display () function");
}
void  print(char *s)
{
 printf("\n%s",s);
}
```

**prog2.c** (source file)

```
#include<stdio.h>
#include "prog1.h"  // include header file prog1

void main()
{
    clrscr();
    display();  //  call display() of prog1.h
    print("My Name is SABAB");   // call print()
    getch();
}
```

**Output:**

```
This is prog1 header  file
Welcome to display () function
My Name is SABAB
```

# UNIT-III

# Control Statements

The control statements enable us to specify the order in which the various instructions in a program are to be executed by the computer. They determine the flow of control in a program.

There are four types of control statements in C. They are:

- Sequence Control Statements
- Decision Control Statements or Conditional Statement.
- Case Control Statement
- Repetition or Loop Control Statements

Sequence Control statements ensure that the instructions in the program are executed in the same order in which they appear in the program.
Decision and Case Control statements allow the computer to take decision as to which statement is to be executed next.

The Loop Control statement helps the computer to execute a group of statements repeatedly.

## ***Conditional Statement***

C has two major decision-making statements.

- **If_else statement**
- **Switch statement**

## *If - else Statement*

The if - else statement is a powerful decision-making tool. It allows the computer to evaluate the expression. Depending on whether the value of expression is "True" or "False" certain group of statements are executed.

The syntax of if else statement is:

```
if (condition is true)
    statement 1;
 else
    statement 2; .
```

The **condition** following the keyword is always enclosed in parenthesis. If the condition is true, statement in then part is executed, that is, statement 1 otherwise statement 2 in else part is executed.

**Example:**

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

```
void main()
   {
     int N;
    clrscr();
    printf("Enter the Number :");
    scanf("%d",&N);

    if(N%2 == 0)
        printf(" Given Number %d is EVEN ",N);
   else
        printf(" Given Number %d is ODD ",N);

    getch();
   }
```

Output :

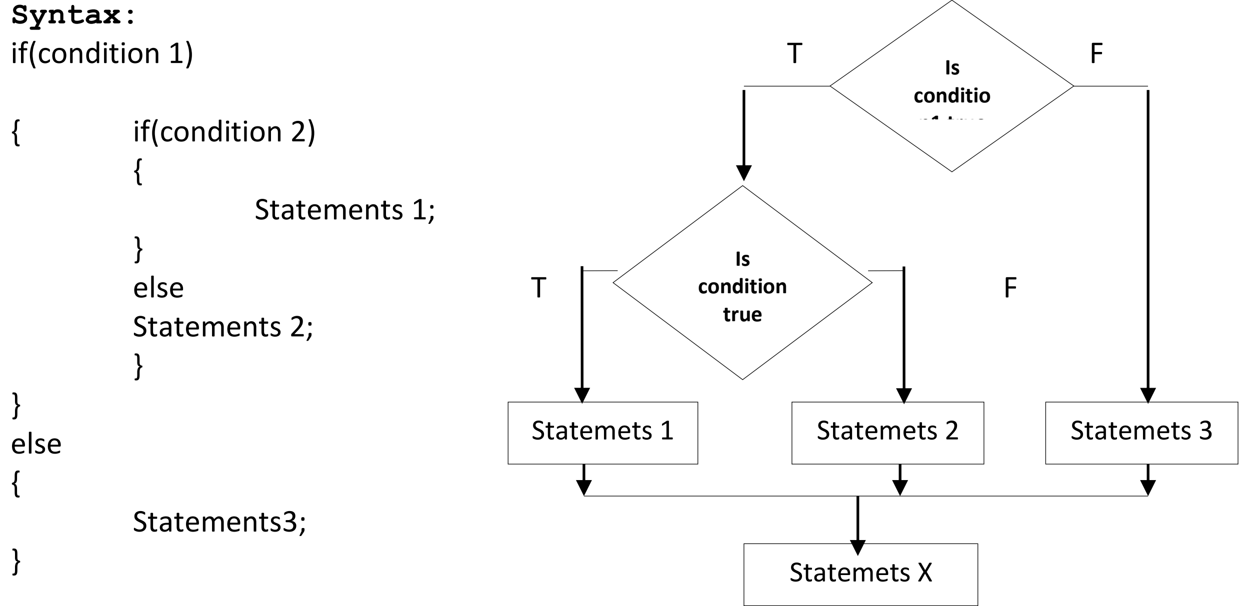
```
Enter the Number: 5
Given Number 5 is ODD
```

## *Nested If-Else Statement:*

It is a conditional statement which is used when we want to check more than 1 conditions at a time in a same program. The conditions are executed from top to bottom checking each condition whether it meets the conditional criteria or not. If it found the condition is true then it executes the block of associated statements of true part else it goes to next condition to execute.

**Syntax:**

```
if(condition 1)

{           if(condition 2)
            {
                        Statements 1;
            }
            else
            Statements 2;
            }
}
else
{
            Statements3;
}
```




In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and again checks the next condition. If it is true then it executes the block of statements associated with it else executes else part.

```
#include <stdio.h>
#include <conio.h>
void main()
{		int no;
		clrscr();
		printf("\n Enter Number :");
		scanf("%d",&no);
		if(no>0)
		{
				printf("\n\n Number is greater than 0 !");
		}
		else
		{
				if(no==0)
				{
						printf("\n\n It is 0 !");
				}
				else
				{
						printf("Number is less than 0 !");
				}
		}
		getch();
}
```

**Output :**

```
Enter Number : 0

It is 0!
```

## *else if Ladder*

In C programming language the else if ladder is a way of putting multiple ifs together when multipath decisions are involved. It is a one of the types of decision making and branching statements. A multipath decision is a chain of if's in which the statement associated with each else is an if. The general form of else if ladder is as follows -

**if** ( condition 1)

{

statement - 1;

}

**else if** (condtion 2)

{

```
            statement - 2;
        }
            else if ( condition n)
            {
                statement - n;
            }
                else
                {
                    default statment;
                }
statement-x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top of the ladder to downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the default statement will be executed.

**Example :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int no;
        clrscr();
        printf("\n Enter Number :");
        scanf("%d",&no);
        if(no>0)
        {
                printf("\n Number is greater than 0 !");
        }
        else if(no==0)
         {
                printf("\n It is 0 !");
        }
        else
        {
                printf("Number is less than 0 !");
        }
        getch();
}
```

**Output :**

Enter Number : 5

Number is greater than 0 !

## *Switch case Statement :*

This is a multiple or multiway branching decision making statement.

When we use nested if-else statement to check more than 1 conditions then the complexity of a program increases in case of a lot of conditions. Thus, the program is difficult to read and maintain. So to overcome this problem, C provides 'switch case'.
Switch case checks the value of an expression against a case values, if condition matches the case values then the control is transferred to that point.

**Syntax:**

```
switch(expression)
{
        case expr1:
                statements;
                break;
        case expr2:
                statements;
                break;

----------------------------
----------------------------
----------------------------
        case exprn:
                statements;
                break;
        default:
                statements;
}
```

In above syntax, switch, case, break are keywords.
expr1, expr2 are known as 'case labels.'
Statements inside case expression need not to be closed in braces.
break statement causes an exit from switch statement.
default case is optional case. When neither any match found, it executes.

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

```
void main()
{
        int no;
        clrscr();
        printf("\n Enter any number from 1 to 3 :");
        scanf("%d",&no);
        switch(no)
        {
                case 1:
                        printf("\n\n It is 1 !");
                        break;
                case 2:
                        printf("\n\n It is 2 !");
                        break;
                case 3:
                        printf("\n\n It is 3 !");
                        break;
                default:
                        printf("\n\n Invalid number !");
        }
        getch();
}
```

**Output :**

```
Enter any number from 1 to 3 : 3

It is 3 !
```

### ** Rules for declaring switch case :*

- The case label should be integer or character constant.
- Each compound statement of a switch case should contain break statement to exit from case.
- Case labels must end with (:) colon.

### ** Advantages of switch case :*

- Easy to use.
- Easy to find out errors.
- Debugging is made easy in switch case.
- Complexity of a program is minimized.

## ***Looping Statements / Iterative Statements :***

A ‘ loop’ is a part of code of a program which is executed repeatedly.
A loop is used using condition. The repetition is done until condition becomes condition true.
A loop declaration and execution can be done in following ways.

- Initialize loop with declaring a variable.
- Check condition to start a loop
- Executing statements inside loop.
- Increment or decrement of value of a variable.

## *** Types of looping statements:**

Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as : Before loop and after loop. So, there are 2(two) types of looping statements.

- Entry controlled loop
- Exit controlled loop

**1. Entry controlled loop:**
In such type of loop, the test condition is checked first before the loop is executed.
Some common examples of this looping statements are :

- while loop
- for loop

**2. Exit controlled loop :**
In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop executed atleat one time compulsarily.
Some common example of this looping statement is :

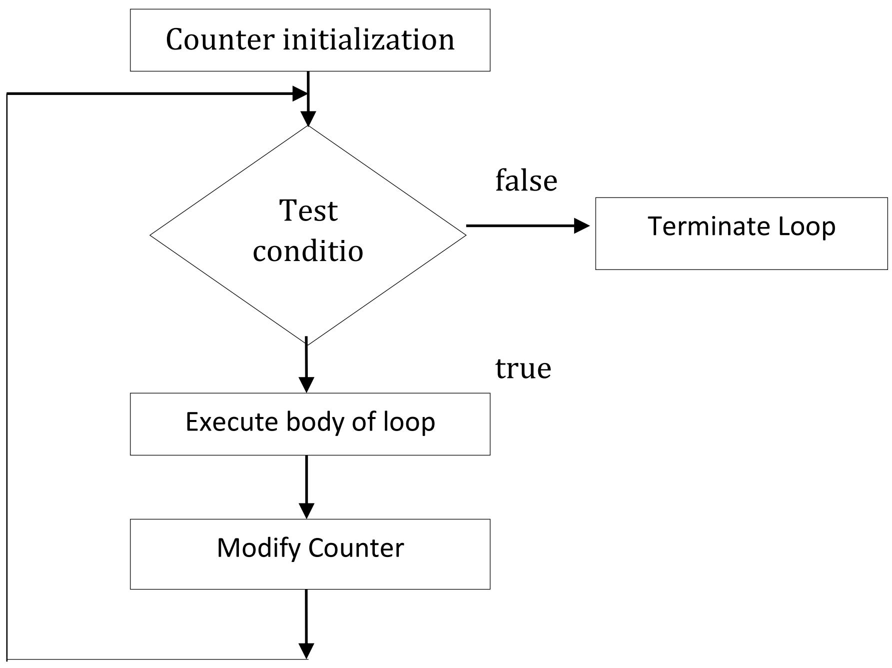
- do-while loop

## For *loop* :

This is an entry controlled looping statement.
In this loop structure, more than one variable can be initialized. One of the most important feature of this loop is that the three actions can be taken at a time like variable initialization, condition checking and increment/decrement. The for loop can be more concise and flexible than that of while and do-while loops.
**Syntax:**

```
for(counter initialization ; test-condition ; modify counter)
{
        statements;
}
```

In above syntax, the given three expressions are separated by ';' (Semicolon)




**Features :**

- More concise
- Easy to use
- Highly flexible
- More than one variable can be initialized.
- More than one increments can be applied.
- More than two conditions can be used.

Example :

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int a;
   clrscr();
   for(i=0; i<5; i++)
   {
            printf("MCMT !\t");  // 5 times
   }
   getch();
}
```

**Output :**

MCMT MCMT MCMT MCMT MCMT

**More About for Loop**

The for loop in C has several capabilities that are not found in other loop constructs. More than one variable can be initialized at a time in the for statement.

The Statement p = 1;
for (n = 0; n < 17; + + n)

can be rewritten as for (p = 1, n = 0; n < 17; ++ n)

The increment section may also have more than one part as given in the following

example:

```
for (n = 1, m = 50; n < = m; n = n+1, m = m-j)
        {
          p = m/n;
         printf ("%d %d %d\n", n, m, p):
        }
```

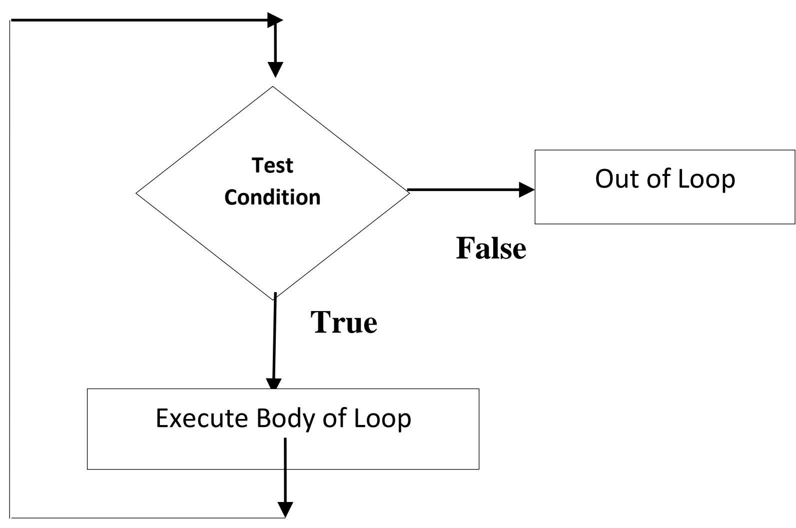
## While loop :

This is an entry controlled looping statement. It is used to repeat a block of statements until condition becomes true.

**Syntax:**

```
while(condition)
{
            statements;
            increment/decrement;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it. At the end of loop increment or decrement is done to change in variable value. This process continues until test condition satisfies.



**Program :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int a;
  clrscr();
  a=1;
  while(a<=5)
  {
        printf("MCMT \t");
        a+=1     // i.e. a = a + 1
  }
  getch();
}
```

**Output :** MCMT MCMT MCMT MCMT MCMT

## Do-While loop :

This is an exit controlled looping statement.

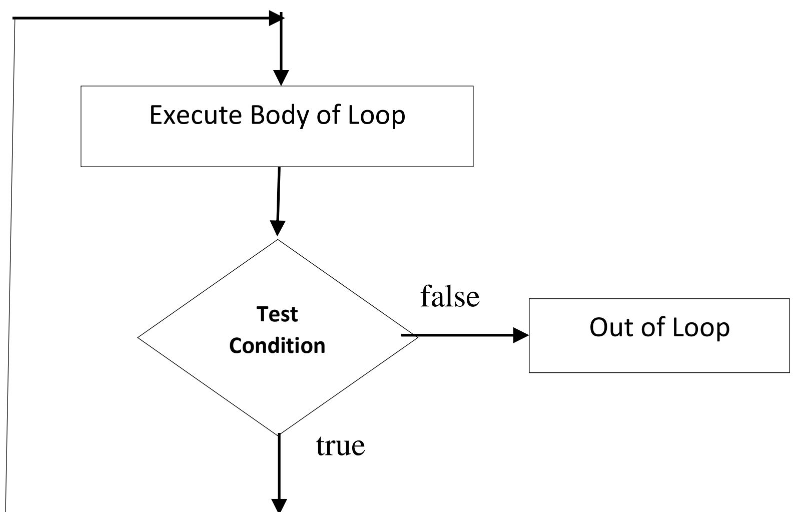
Sometimes, there is need to execute a block of statements first then to check condition. At that time such type of a loop is used. In this, block of statements are executed first and then condition is checked.

**Syntax:**

```
do
{
        statements;
        (increment/decrement);
}while(condition);
```

In above syntax, the first the block of statements are executed. At the end of loop, while statement is executed. If the resultant condition is true then program control goes to evaluate the body of a loop once again. This process continues till condition becomes true. When it becomes false, then the loop terminates.

**Note: The while statement should be terminated with ; (semicolon).**




**Program :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int a;
  clrscr();
  a=1;
  do
  {
        printf("MCMT\t");  // 5 times
        a+=1;     // i.e. a = a + 1
  }while(a<=5);
  getch();
}
```

**Output :** MCMT MCMT MCMT MCMT MCMT

## Infinite loop :

A looping process, in general, includes the following four steps:

- Setting of a counter.
- Execution of the statements in the loop.
- Testing of a condition for loop execution.
- Incrementing the counter.

The test condition eventually transfers the control out of the loop. In case, due1 to some reasons, if-it does not do so, the control sets up an infinite loop and the loop body is

executed over and over again. Such infinite loops should be avoided. Ctrl+C or Ctrl+Break are used to terminate the program caught in an infinite loop. Two examples of infinite loop are given below:

**Example :**

```
#include<stdio.h>
 void main()
        {
                int i=1;
                while(i<=10)
                {
                  Printf(" i= %d\n",i);
                }
        }
```

This program will never terminate as variable i will always be less than 10. To get the loop terminated, an increment operation (i + +) will be required in the loop.

## Nested Loops :

Loops within loops are called nested loops. An overview of nested while, for and do .. while loops is given below:

### *Nested while :*

It is required when multiple conditions are to be tested. The syntax of nested while is:

```
while (condition 1)
   {    ----------


        while (condition 2)
           {    ----------


                while (condition n)
                   {    ----------


                   }
           }
   }
```

**For example:**

```
#include<stdio.h>
#include<conio.h>
void main()
{    int  i = 1, N;
```

```
    clrscr();
   while( i<= 5 )
    {
        N=1;
        while( N<=5)
          {
                printf("  %d  " , N);
          }
        printf( " \n");
     }
}
```

**Ouput :**

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

## *Nested for :*

It is used when multiple set of iterations are required. The syntax of nested for is:

```
for (   ;    ;   )
   {   ----------
       ----------------
       ---------------

         for  (    ;    ;   )
            {   ----------
                 ----------
                 for  (    ;    ;   )
                    {   ----------
                        ----------------
                        ---------------
                    }
            }
   }
```

**For example:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int  i , N;
   clrscr();
  for(  i=1 ; i<= 5 ; i++ )
   {
     for(  N=1 ;N<= 5 ; N++ )
        {
           printf("  %d  " , N);
     }
printf( " \n");
   }
 }
```

**Ouput :**

```
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
```

## The break Statement

The break statement is used to terminate loops or to exit from a switch. When break is encountered inside any C loop, control automatically passes to the first statement after the loop. It can be used within a while, a do-while, a for loop or a switch statement. The break statement is written simply as break; without any embedded expression of statements.

Sometimes, it is necessary to exit immediately from a loop as soon as the condition is satisfied.

The statements after break statement are skipped.

**Syntax :**

```
break;
```

**Figure :**

```
while (condition)
{
   _ _ _
   _ _ _ _
     break  ;
   _ _ _ _
   _ _ _
}
```

**Example :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int i;
   clrscr();
   for(i=1; i<=20 ; i++)
   {
          if(i>5)
          break;
          printf("%d",i);  // 5 times only
   }
printf(" \nOut of loop");
   getch();
}
```

Output :

12345
Out of loop

## Continue Statement :

Sometimes, it is required to skip a part of a body of loop under specific conditions. So, C supports 'continue' statement to overcome this anomaly.

The working structure of 'continue' is similar as that of that break statement but difference is that it cannot terminate the loop. It causes the loop to be continued with next iteration after skipping statements in between. Continue statement simply skipps statements and continues next iteration.

**Syntax :**

```
continue;
```

**Figure :**

```
while (condition) <--+
{                    |
   _ _ _             |
   _ _ _ _           |
   continue; --------+
   _ _ _ _
   _ _ _
}
```

**Example:**

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int i;
        clrscr();
        for(i=1; i<=10; i++)
        {
                if( i>=6 && i <=8 )
                continue;
                printf("\t%d",i);  // 6 to 8 is omitted
        }
        getch();
}
```

**Output :**

```
1   2   3   4   5   9   10
```

## The exit( ) Function

The exit() function is used to terminate the execution of 'C' program. It is a standard library function and uses header file stdlib.h.
The general form of exit( ) function is

**exit (int status);**

The difference between break and exit( ) is that former terminates the execution of loop in with it is written while exit( ) terminates the execution of program itself.

The status (in the general form of exit) is a value returned to the operation system after the termination. of the program.

The value zero “0” indicates that the termination is normal while value one “1” (Non-Zero) indicates
different types of errors.

**Example :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int i;
   clrscr();
   for(i=1;  ; i++)
   {
           if(i>5)
           exit(0);
           printf("%d",i);  // 5 times only
   }
printf(“ \nOut of loop”);   // control will not reached here
   getch();
}
```

**Output :**

    12345

# The goto Statement

C supports the **goto** statement to branch unconditionally from one point to another in the program. A goto statement breaks the normal sequential execution of 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.

It is a well known as "**jumping statement**". It is useful to provide branching within a loop.
When the loops are deeply nested at that if an error occurs then it is difficult to get exited from such loops. Simple break statement cannot work here properly. In this situation, goto statement is used.

The general forms of goto and label statements are:

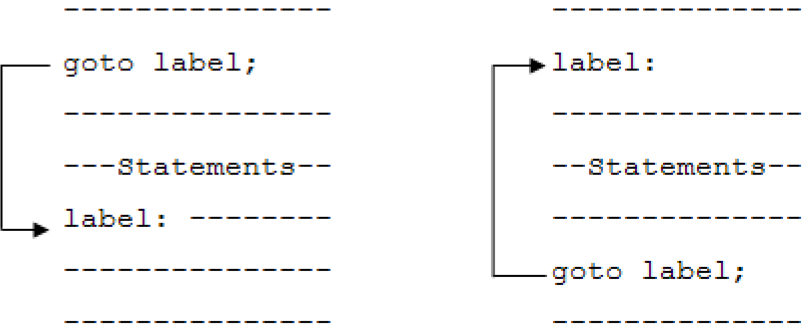



**Figure :**

```
while (condition)
{
    for ( ;  ;  ;)
    {
       _ _ _
       _ _ _ _
       goto err;
       _ _ _ _
       _ _ _
    }
  err:
}
```

**Example :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int i=1, j;
   clrscr();
   while(i<=3)
   {
            for(j=1; j<=3; j++)
            {
                     printf(" I=%d \t J=%d \n", i , j);
                     if(j==2)
                     goto stop;
            }
            i = i + 1;
   }

   stop:   printf("\n Exited !");
   getch();
}
```

**Output :**

```
I=1        J=1
I=1        J=2

Exited!
```

# UNIT-IV

# Basics of Programming

## Computer programming

Designing and writing computer programs, or sequences of instructions to be executed by a computer. A computer is able to perform useful tasks only by executing computer programs. A programming language or a computer language is a specialized language for expressing the instructions in a computer program.

## Problem Definition

It is the first and most important step in determining the information needs.
Information needs of software can be determined by-

1) Studying the existing System
2) Using questionnaires
3) Suggesting interpretation of information needs with users

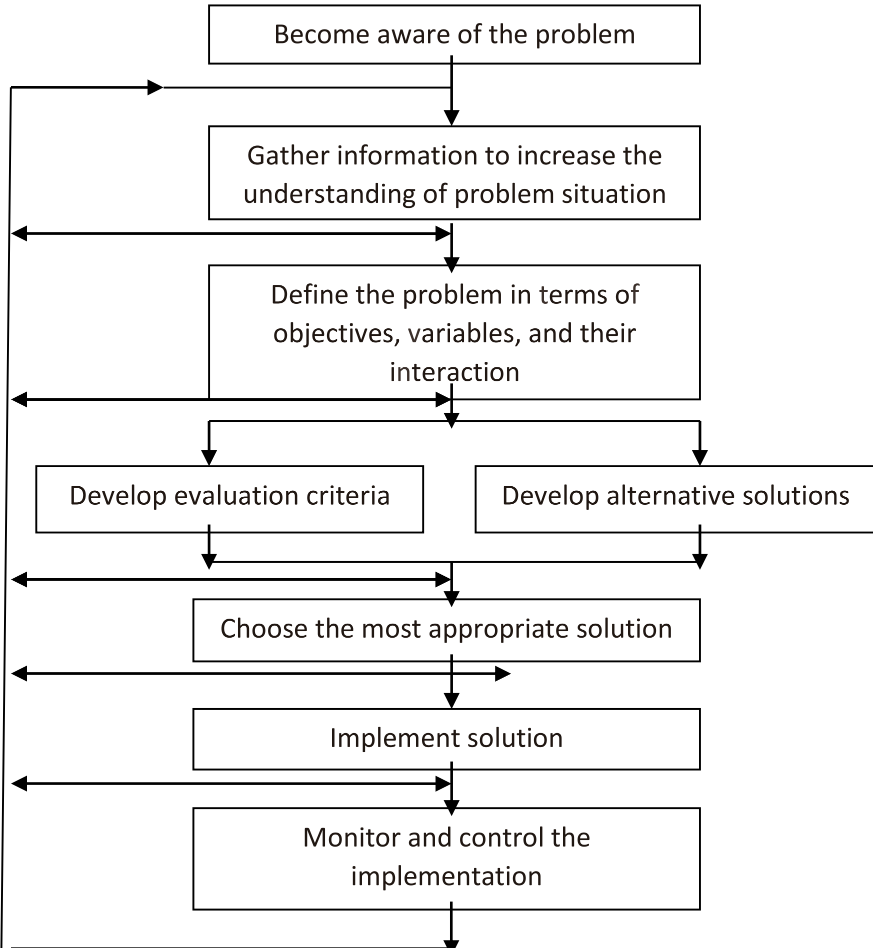
Using any of these steps first of all the requirements of determines.
This phase in the product life cycle encompasses two different activities: Concept Development and Requirements Identification. In Concept Development, we gather and refine ideas and set cohesive product goals. Requirements Identification then further clarifies the goals by specifying the needs and functionality of the product.

## Problem solving

In this stage, the programmer gains a full understanding of the problem that the computer program under development is supposed to solve, and devises a step-by step procedure (an algorithm) that, when followed, will solve the problem.

Problems are the undesirable situations that prevent any software from fully achieving its objectives. Problem dearly defined in terms of goals and objectives helps largely in problem solving. Steps involved in problem solving methodology are shown in the figure.

## Steps in Problem Solving




## Some problem-solving techniques

- **Trial-and-error (also called guess and check)**
- **Brainstorming**
- **divide and conquer**

- ## Trial and Error To Solve Problems

Some complex problems can be solved by a technique that is called trial and error. Trial and error is typically good for problems where you have multiple chances to get the correct solution. However, this is not a good technique for problems that don't give you multiple chances to find a solution.

An example of situations where you wouldn’t want to use trial and error are diffusing a bomb or performing an operation on a patient. In these situations, making an error can lead to disaster. Trial and error is used best when it is applied to situations that give your large amounts of time and safety to come up with a solution.

In addition to this, trial and error is also a great way to gain knowledge. Basically, a person that uses the trial and error method will try to a method to see if it is a good solution. If it is not a good solution, they try another option. If the method works, the person using it has acquired the correct solution to a problem.

- ## Brainstorming

Brainstorming as a technique was first introduced by Alex Osborne in the 1930s. It is a method used in groups in order to support creative problem-solving, the generation of new ideas and greater acceptance of proposed solutions.

Brainstorming works by focusing on a problem, and then intentionally coming up with as many solutions as possible and by pushing the ideas as far as possible. One of the reasons it is so effective is that the brainstormers not only come up with new ideas in a session, but also inspire from associations with other people's ideas by developing and refining them.

### Benefits of brainstorming

The benefits of a well-organized brainstorming session are numerous. They include:

(a) Solutions can be found rapidly and economically;

(b) Results and ways of problem-solving that are new and unexpected

(c) A wider picture of the problem or issue can be obtained;

(d) The atmosphere within the team is more open;

(e) The team shares responsibility for the problem;

(f) Responsibility for the outcome is shared;

(g) The implementation process is facilitated by the fact that staff shared in the

decision-making process.

- ## Divide-and-conquer

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

divide-and-conquer paradigm consists of following major phases:

Breaking the problem into several sub-problems that are similar to the original problem but smaller in size, Solve the sub-problem recursively (successively and independently), and then Combine these solutions to subproblems to create a solution to the original problem.

# Pseudocode

**Pseudocode** is an informal high-level description of the operating principle of a computer program or other algorithm.

Pseudocode (pronounced SOO-doh-kohd) is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language. Pseudocode is sometimes used as a detailed step in the process of developing a program. It allows designers or lead programmers to express the design in great detail and provides programmers a detailed template for the next step of writing code in a specific programming language.

three basic constructs for flow of control are sufficient to implement any "proper" algorithm.

- **SEQUENCE** is a linear progression where one task is performed sequentially after another.
- **WHILE** is a loop (repetition) with a simple conditional test at its beginning.
- **IF-THEN-ELSE** is a decision (selection) in which a choice is made between two alternative courses of action.

**1-SEQUENCE**

Sequential control is indicated by writing one action after another, each action on a line by itself, and all actions aligned with the same indent. The actions are performed in the sequence (top to bottom) that they are written.

**Example**

READ height of rectangle
READ width of rectangle
COMPUTE area as height times width

**2-IF-THEN-ELSE**

Binary choice on a given Boolean condition is indicated by the use of four keywords: IF, THEN, ELSE, and ENDIF. The general form is:

IF condition THEN
    sequence 1
ELSE
    sequence 2
ENDIF

The ELSE keyword and "sequence 2" are optional. If the condition is true, sequence 1 is performed, otherwise sequence 2 is performed.

### Example

```
IF HoursWorked > NormalMax THEN
      Display overtime message
ELSE
      Display regular time message
ENDIF
```

### 3- WHILE

The WHILE construct is used to specify a loop with a test at the top. The beginning and ending of the loop are indicated by two keywords WHILE and ENDWHILE. The general form is:

WHILE condition
        sequence
ENDWHILE

The loop is entered only if the condition is true. The "sequence" is performed for each iteration. At the conclusion of each iteration, the condition is evaluated and the loop continues as long as the condition is true.

Example

```
get a number
set our initial count to 0
while our number is greater than 1
      divide the number by 2
      increase our count by 1
end
```

# Big O notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

below are some common orders of growth along with descriptions and examples where possible.

## *O(1)*

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
void func()
{
Print “Hello!”
}
```

## *O(N)*

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

The example below also demonstrates how Big O favors the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(char[] elements, string value)
{
   foreach (var  x in elements)
   {
      if (x == value) return true;
   }
   return false;
}
```

## $O(N^2)$

$O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

```
bool ContainsDuplicates(IList<string> elements)
{
   for (var outer = 0; outer < elements.Count; outer++)
   {
      for (var inner = 0; inner < elements.Count; inner++)
      {
         // Don't compare with self
         if (outer == inner) continue;

         if (elements[outer] == elements[inner]) return true;
      }
   }

   return false;
}
```

## $O(2^N)$

$O(2^N)$ denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an $O(2^N)$ function is exponential - starting off very shallow, then rising meteorically. An example of an $O(2^N)$ function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
   if (number <= 1)
        return number;
    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

### Logarithm

The most common attributes of logarithmic running-time function are that:

- the choice of the next element on which to perform some action is one of several possibilities, and only one will need to be chosen.

  Or

  the elements on which the action is performed are digits of n

- **O(log n):** Given a person's name, find the phone number by picking a random point about halfway through the part of the book you haven't searched yet, then checking to see whether the person's name is at that point. Then repeat the process about halfway through the part of the book where the person's name lies. (This is a binary search for a person's name.

- **O(n):** Find all people whose phone numbers contain the digit "5".
- **O(n):** Given a phone number, find the person or business with that number.
- **O(n log n):** There was a mix-up at the printer's office, and our phone book had all its pages inserted in a random order. Fix the ordering so that it's correct by looking at the first name on each page and then putting that page in the appropriate spot in a new, empty phone book.

# Algorithms

The concept of an algorithm is one of the basic concepts in mathematics. An algorithm is a finite set of rules, which gives a sequence of operations to solve a specific problem.

**Example:** [Making a Good Cup of Tea]

Step 1. Take a beaker and place it on a burner (stove)

Step 2. Add ½ Cup water and ½ Cup Milk

Step 3. Heat it to 100deg. BOILING point

Step 4. Add Tea powder or leaves.

Step 5. Let boil for some time.

Step 6. Add one tea spoon sugar and Extra Items which u want.

Step 7. Let boils for some time

Step 8. Filter it and Serve in a Cup (your tea is ready .......)

The word "algorithm" originates from the Arabic word algorism which has been derived from the name of a famous Arabic mathematician Abu Jajar Mohammed ibn Musa AI Khwarizrni (AD 825) who was the first to suggest a mechanical method to add two numbers represented in the Hindu numeral system.

## Properties of an Algorithm

Algorithm is a step-by-step problem solving procedure that can be Carried out by a computer. The essential properties of an algorithm are:

**Finiteness**: An algorithm should terminate after a finite number of steps, that is, when it is mechanically executed, it should come to a stop after executing a finite number of. Assignment, decision and repetitive steps.

**Definiteness**: An algorithm should be simple. Each step of the algorithm should be precisely defined, that is, the steps must be unambiguous so that the computer understands them properly.

**Generality**: An algorithm should be complete in itself, that is, it should be able-to solve all problems of a particular type.

**Effectiveness**: All the operations used in the algorithm should be basic and capable of being performed mechanically.

**Input-output**: An algorithm should take certain precise inputs, or initial data, and the outputs should be generated in the intermediate as well as the steps of the algorithm.

# Algorithm Logic

Program analysts have found that algorithms developed using three basic components are easier to follow. These three components which are considered as standard units to control the flow of information processing and are used to construct algorithms are:

## Sequential Flow

In a sequential component, steps are taken in an explicitly prescribed sequence.

For example,

INPUT R, H, D

G = R*H

P = G - D

Update record

Print cheque

## Conditional Flow

Algorithm involves a decision to be made based on a condition, the flow is said to be Conditional, To make a decision, a condition is tested. If the condition is true, then one is taken; if false, then the other. For example,

INPUT R, H, D

G = R*H

P = G - D

If P is positive,

then print a check.

Upate record.

## Repetitive Flow

The Other basic-variation in algorithm flow deals with repetition of steps. In this case, if the Condition is true, then the steps in the procedure are taken, after which flow is back to the condition for a possible repetition. As long as the condition remains true, this path is followed.

Step 1 A=1

Step 2 Print A

Step 3 A=A+1

Step 4 If A < = 10 goto Step 2

Step 5 End

# Examples

**Example 1: Addition and multiplication of two numbers**

Step 1 Input values for A and B

Step 2 Add B to A and store in 'SUM

Step 3 Multiply B with A and store in MUL

Step 4 Display value of SUM and MUL

Step 5 End.

**Example 2: Check for even or odd numbers**

Step 1 Accept value in variable NUM

Step 2 Divide NUM by 2 and store remainder in REM

Step 3 If REM is zero

Display "Number is even"

Else

Display "Number is odd"

Step 4 : End.

**Example 3: Swapping two variables with the help of the third variable**

Step 1 Input values in A and B.

Step 2 Let variable T = A

Step 3 Let A = B

Step 4 Let B. = T

Step 5 Display A and

Step 6 End.

**Example 4: Swapping two variables without using the third variable**

Step 1 Input values ih A and B.

Step 2 LatA = A+B

Step 3 Let B = A-B

Step 4 Let A = A-B

Step 5 Display A and B

Step 6 End.

**Example 5: To calculate the percentage of marks obtained by the student in an examination. the problem, maximum marks and marks obtained are given. The required result is percentage of marks and the formula used is:**

$$\% \text{ of marks} = \frac{\text{marks obtained}}{\text{max. marks}} * 100$$

Step 1 Read name, marks obtained & maxes. Marks.

Step 2 Divide marks obtained by max. marks and store it in Per:

Step 3 multiply Per. by 100 to get percentage.

Step 4 Writte name and .percentage.

Step 5 Stop.

**Example 6: Check for prime number**

Step 1 Input a value in NUM
Step 2 If NUM is 0, 1, 2 or 3 go to step 8
Step 3 Let variable E = NUM/2 and I = '2
Step 4 Store remainder of NUM/I in REM
Step 5 If REM = 0 go to step 9.
Step 6 I = 1+1
Step 7 If I < = E go to step 4
Step 8 Display "Number is prime": go to step 10
Step 9 Display "Number is Not prime"
Step 10 End

**Example 7: Summation of a set of numbers**

Step 1 Input total number of variables, for summation in N
Step 2 Let Sum = 0, I = 1
Step 3 Input a number in NUM
Step 4 SUM = SUM + NUM
Step 5 I = I + 1
Step 6 If I <= N go to step 3
Step 7 Display SUM
Step 8 End.

**Example 8: Display even numbers from 0-50**

Step 1 Let I = 0
Step 2 If remainder of I/2 is not zero go to step 4
Step 3 Display I: go to step 6
Step 4 I = I + 1
Step 5 If I <= 50 go to step 2
Step 6 End.

**Example 9: Factorial computation**

Step 1 Input value in N

Step 2 Let FACT = 1

Step 3 FACT = FACT * N

Step 4 N = N-1

Step 5 If N > 1 go to step 3

Step 6 Display value of FACT

Step 7 End.

**Example 10: Fibonacci series generation**

Step 1 Input the last term of series in N

Step 2 Let A=0, B=1 and X=3

Step 3 Display A and B

Step 4 Let C = A+B

Step 5 Display C

Step 6 Let A = B and B = C

Step 7 X = X+ 1

Step 8 if X<= N go to step 4

Step 9 End.

**Example 11: Reversing digits of an integer**

Step 1 Input a value in NUM

Step 2 Let REV = 0,

Step 3 Store. remainder of NUM/10 in Rem

Step 4 REV = REV * 10 + REM

Step 5 NUM = (integer) NUM/10

Step 6 If NUM > 0 go to step 3.

Step 7 Display REV

Step 8 End.

**Example 12: Conversion of decimal base number to binary**

Step 1 Input number in DECI

Step 2 Let BBASE = 0 and I = 0

Step 3 Store remainder of DECI/2 in REM

Step 4 BBASE = BBASE + REM * (10 power I)

Step 5 I = I + 1

Step 6 DECI = (Integer) DECI/2

Step 7 If DECI != 0 go to step 3

Step 8 Display BBASE

Step 9 End.

# FLOWCHART

A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts facilitate communication between programmers and business people.

Once the flowchart is drawn, it becomes easy to write the program in any high level language.

It is a type of diagram that represents an algorithm or process.

## Advantages of flowchart:-

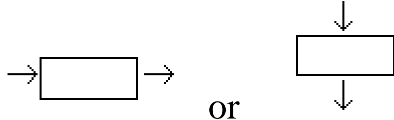
1. It provides an easy way of communication because any other person besides the programmer can understand the way they are represented.
2. It represents the data flow.
3. It provides a clear overview of the entire program and problem and solution.
4. It checks the accuracy in logic flow.
5. It documents the steps followed in an algorithm.
6. It provides the facility for coding.

### Symbols for drawing Flowchart

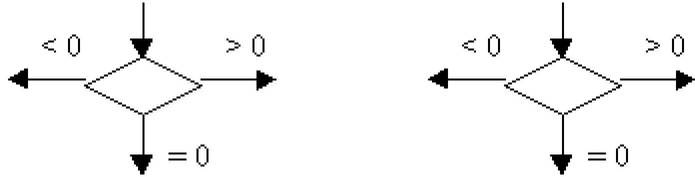
| Symbol | Name |
|---|---|
| | START / END |
| | INPUT / OUTPUT BOX |
| | PROCESSING BOX |
| | DECISION BOX |
| | FLOW-LINE |
| | CONNECTOR |

The following are some guidelines in flowcharting:

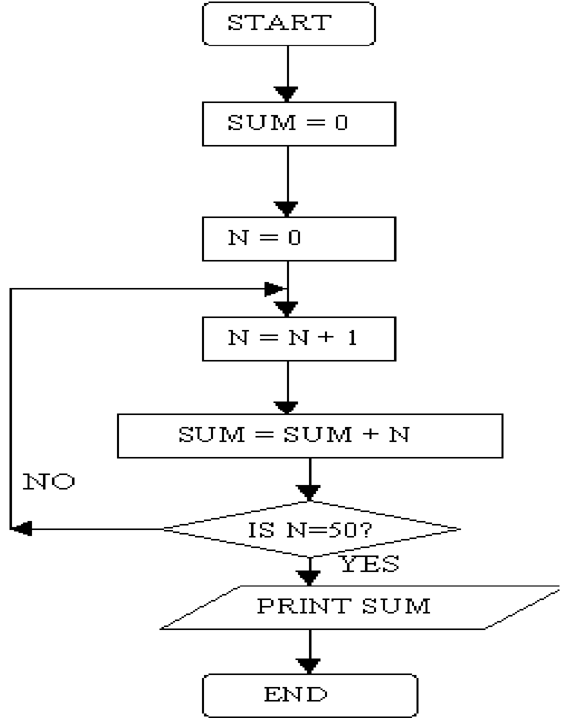
a. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
b. The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
c. The usual direction of the flow of a procedure or system is from left to right or top to bottom.
d. Only one flow line should come out from a process symbol.



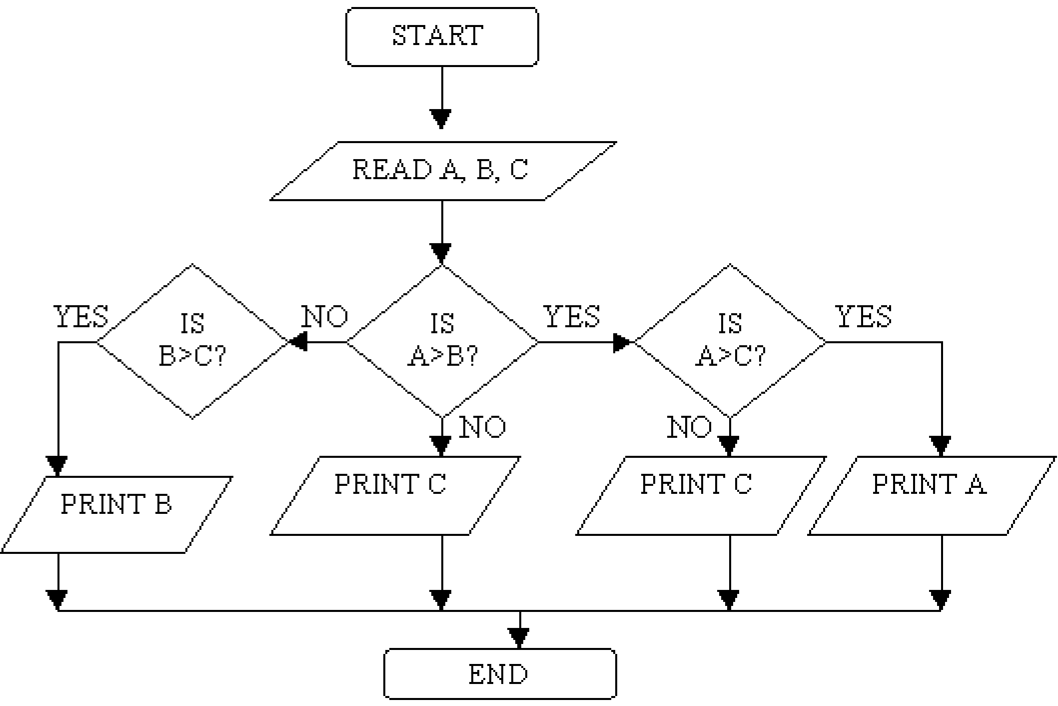

e. Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.



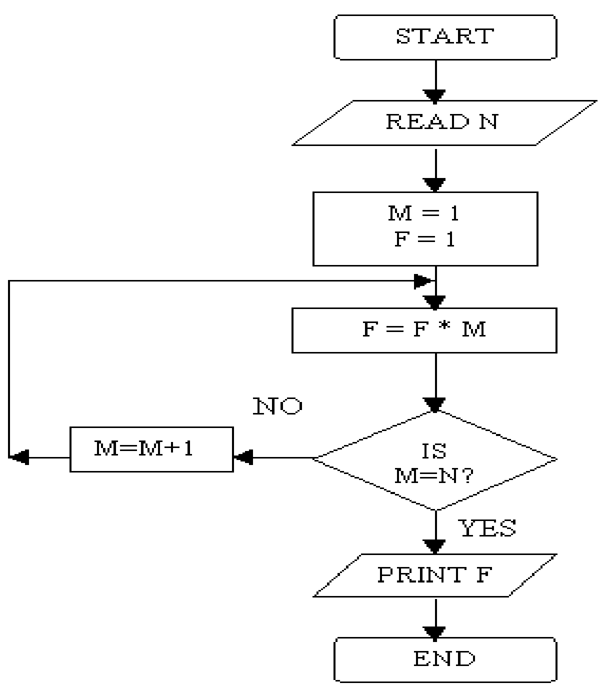

**Example 1**: ***Draw a flowchart to find the sum of first 50 natural numbers.***



**Example 2:** Draw a flowchart to find the largest of three numbers A,B, and C.




**Example 3:** Draw a flowchart for computing factorial N (N!)
Where N! = 1 ´ 2 ´ 3 ´ …… N.



# Program

Program is a set of computer understandable instructions to solve a computational problem. The specification of the sequence of computational steps in a particular programming language is termed as a program.

# Programming

The task of developing programs is called programming and the person engaged in programming activity is called a programmer.

# System Design Techniques

## TOP Down Design

The top down design approach is based on the fact that large problems become more manageable if they are divided into a number of smaller and simpler tasks which can be tackled separately.

Top down design approach is performed in a special way. The main program is written first. It is tested before sub programs are written. After the main program is written and checked, each module is written and tested in turn.

## Bottom-Up Design

The pure bottom up approach is generally not recommended because it is difficult to anticipate which low-level subroutines will be needed for any particular program. In the bottom up approach it is usually assumed that the basic routines created will be general enough to be used more than once. Using the subroutines, to construct a program, save repeating the same lines of code by re-using it.

# Programming Techniques

## Linear programming

Linear program is a method for straight forward programming in a sequential manner. This type of programming does not involve any decision making. General model of these linear programs is:

- Read a data value.
- Compute an intermediate result.
- Use the intermediate result to compute the desired answer.
- Print the answer.
- Stop.

## Structured programming

One of the most versatile properties of a digital computer is that it can make a "decision", thus creating a branching point. If branching and looping can be used, then much more complex iterative algorithms can be written, which in-turn results in more complex programs. This technique for writing such programs are referred to as Structured Programming.

## Advantages of Structured programming

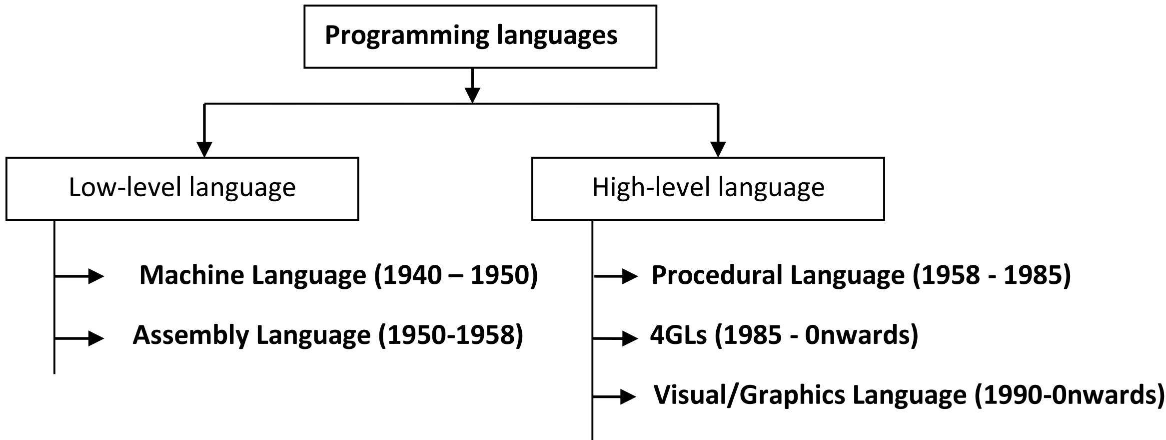
- Decreases the complexity of the program by breaking it down into smaller logical units. .
- Allow several programmers to code simultaneously.
- Allows common functions to be written once and then used in all the programs needing it.
- Decreases debugging time, because modules make it easier to isolate errors.
- Modification to single modules do not affect the rest of the program.

# Computer Programming Languages

Computer does not understand natural language because it is an electronic device which works on the instructions provided by the user. It is required to provide the instructions in some computer understandable language. Such a computer understandable language is known as programming language.

# Types of Programming Languages

Types of programming languages can be easily explained by the following chart.



## Machine-Level Language (1st Generation)

Computer can understand binary codes (1, 0) only: So the instructions given to the computer can only be in 1 or o. The language which contains binary codes is called machine-level language.

### Advantages

- Machine-level instructions are directly executable.
- Machine-level language makes most efficient use of computer system resources like storage, and register.
- Machine language instructions can be used to manipulate individual bits.

### Disadvantages

- As machine-Level languages are device dependent, the programs are not portable from one computer to another.
- Programming in machine language usually results in poor programmer productivity.
- Programs in machine language are more error prone and difficult to debug.
- Computer storage locations must be addressed directly, not symbolically.

## Assembly Language (2nd Generation)

●

Assembly languages are' also known as second-generation languages, These Languages substitute alphabetic symbols for the binary codes of machine language.

### Advantages

- Assembly language is easier to use than machine language.
- An assembler is useful for detecting programming errors.
- Programmers do not have to know the absolute addresses of data items.
- Assembly languages encourage modular programming.

### Disadvantages

- Assembly Language programs are not directly executable.
- Assembly languages are machine dependent and, therefore, not portable from one machine to another.
- Programming in assembly language requires a higher level of programming skill.

## High-Level Languages (3rd Generation)

These are the third-generation languages. These are procedure-oriented languages and are machine independent Programs are written in English like statements. As high-level languages are not directly executable, translators (compilers and interpreters) are used to convert them in machine language equivalent.

### Advantages

- These are easier to learn than assembly language.
- Less time is required to write programs.
- They provide better documentation.
- They are easier to maintain.
- They have an extensive vocabulary.
- Libraries of subroutines can be incorporated and used in many other programs.
- Programs written in high-level languages are easier to debug because translators display all the errors with proper error messages at the time of translation.

### 4GLs (Fourth-Generation Programming Languages )

4GLs were developed in late 1970s and early 1980s. They concentrate on what is to be accomplished rather than how it is to be accomplished. 4GLs facilitate interactive coding in form of an on-screen menu choice to formulate an enquiry or define a task.

## Assembler

An assembler is a program which is used to translate an assembly language program into its machine-level language equivalent. The program in assembly language is termed as source code and its machine language equivalent is called object program.
Once the object program is created, it is transferred into the computer's primary memory using the System Loader for execute program.

## Compiler

Compilers are System program that takes high-level language program (source code) as input and produce machine-level language program (object code) as output. Compiler allocates addresses to all variables and statements. It also tabulates a list of programming errors found during compilation.

## Interpreter

It is also used for translating high-level language program into machine-level language. The main difference between compiler and interpreter is that compiler translates entire program first and then produces the listing of errors. While the interpreters perform line by line translation. As soon as the error is encountered, interpretation process stops. Due to line--line translation, interpreters are slower than compilers. The most well-known interpreter-based language is BASIC.

# UNIT-V

## Basic Programs in C Language

### Addition/Subtraction/Multiplication of Integers

```
#include <stdio.h>

int main()
{
   int first, second, add, subtract, multiply;

   printf("Enter two integers\n");
   scanf("%d%d", &first, &second);

   add = first + second;
   subtract = first - second;
   multiply = first * second;

   printf("Sum = %d\n",add);
   printf("Difference = %d\n",subtract);
   printf("Multiplication = %d\n",multiply);

   return 0;
}
```

### Determine the number is +ive or -ive

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

    printf("Enter a number: ");
    scanf("%lf", &number);

    if (number <= 0.0)
    {
        if (number == 0.0)
            printf("You entered 0.");
        else
            printf("You entered a negative number.");
    }
    else
        printf("You entered a positive number.");
    return 0;
}
```

### Determine number is Even or Odd

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

  printf("Enter an integer\n");
  scanf("%d", &n);

  if (n%2 == 0)
    printf("Even\n");
  else
    printf("Odd\n");

  return 0;
}
```

**Greatest between two numbers**

```
#include <stdio.h>

int main()
{
   int num1, num2;

   /*
    * Reads any two integer values from user
    */
   printf("Enter any two numbers:\n");
   scanf("%d %d", &num1, &num2);

   /*
    * Check if num1 > num2 or not and prints the maximum
    */
   if(n1 > n2)
   {
      printf("%d is maximum", num1);
   }
   else
   {
      printf("%d is maximum", num2);
   }

   return 0;
}
```

**Greatest between three numbers**

```
#include <stdio.h>
int main()
{
   int n1, n2, n3;
```

```
    printf("Enter three numbers: ");
    scanf("%d %d %d", &n1, &n2, &n3);

    if( n1>=n2 && n1>=n3 )
        printf("%d is the largest number.", n1);

    if( n2>=n1 && n2>=n3 )
        printf("%d is the largest number.", n2);

    if( n3>=n1 && n3>=n2 )
        printf("%d is the largest number.", n3);

    return 0;
}
```

**Sum of first n numbers**

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    for(i=1; i <= n; ++i)
    {
        sum += i;   // sum = sum+i;
    }

    printf("Sum = %d",sum);

    return 0;
}
```

**Sum of given n numbers**

```
#include <stdio.h>

int main()
{
   int n, sum = 0, c, value;

   printf("Enter the number of integers you want to add\n");
   scanf("%d", &n);

   printf("Enter %d integers\n",n);

   for (c = 1; c <= n; c++)
   {
      scanf("%d", &value);
```

```
    sum = sum + value;
  }

  printf("Sum of entered integers = %d\n",sum);

  return 0;
}
```

**Digit Reversing**

```
#include <stdio.h>

int main()
{
  int n, reverse = 0;

  printf("Enter a number to reverse\n");
  scanf("%d", &n);

  while (n != 0)
  {
    reverse = reverse * 10;
    reverse = reverse + n%10;
    n       = n/10;
  }

  printf("Reverse of entered number is = %d\n", reverse);

  return 0;
}
```

**Table generation of n**

```
#include <stdio.h>
int main()
{
   int n, i;

   printf("Enter an integer: ");
   scanf("%d",&n);

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

   return 0;
}
```

## Program to calculate $a^b$

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a,b,i,pow;
 clrscr();
 printf("Enter value of a: ");
 scanf("%d",&a);
 printf("Enter value of b: ");
 scanf("%d",&b);
 pow=1;
 for(i=1;i<=b;i++)
 {
  pow=pow*a;
 }
 printf("a(power)b = %d",pow);
 getch();
}
```

**Pascal Triangle**

```
#include <stdio.h>
 long factorial(int);
int main()
{
  int i, n, c;

  printf("Enter the number of rows you wish to see in pascal triangle\n");
  scanf("%d",&n);

  for (i = 0; i < n; i++)
  {
    for (c = 0; c <= (n - i - 2); c++)
      printf(" ");

    for (c = 0 ; c <= i; c++)
      printf("%ld ",factorial(i)/(factorial(c)*factorial(i-c)));

    printf("\n");
  }

  return 0;
}

long factorial(int n)
{
```

```
   int c;
   long result = 1;

   for (c = 1; c <= n; c++)
         result = result*c;

   return result;
}
```

**Swapping in two numbers**

```
#include <stdio.h>
int main()
{
   int x, y, temp;
   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);
   printf("Before Swapping\nx = %d\ny = %d\n",x,y);
   temp = x;
   x    = y;
   y    = temp;
   printf("After Swapping\nx = %d\ny = %d\n",x,y);
   return 0;
}
```

**Swapping in two numbers without using third variables**

```
#include <stdio.h>
int main()
{
   int a, b;
   printf("Enter two integers to swap\n");
   scanf("%d%d", &a, &b);
   a = a + b;
   b = a - b;
   a = a - b;
   printf("a = %d\nb = %d\n",a,b);
   return 0;
}
```

**C program to find nCr and nPr using function**

```
#include <stdio.h>

long factorial(int);
long find_ncr(int, int);
long find_npr(int, int);

int main()
{
   int n, r;
   long ncr, npr;
   printf("Enter the value of n and r\n");
```

```
   scanf("%d%d",&n,&r);
   ncr = find_ncr(n, r);
   npr = find_npr(n, r);
   printf("%dC%d = %ld\n", n, r, ncr);
   printf("%dP%d = %ld\n", n, r, npr);
   return 0;
}
long find_ncr(int n, int r) {
   long result;
   result = factorial(n)/(factorial(r)*factorial(n-r));
   return result;
}

long find_npr(int n, int r) {
   long result;

   result = factorial(n)/factorial(n-r);

   return result;
}

long factorial(int n) {
   int c;
   long result = 1;

   for (c = 1; c <= n; c++)
      result = result*c;

   return result;
}
```

**Integer Division of Numbers**

```
#include <stdio.h>

int main()
{
   int first, second;
   float divide;
   printf("Enter two integers\n");
   scanf("%d%d", &first, &second);
   divide = first / (float)second;   //typecasting
   printf("Division = %.2f\n",divide);
return 0;
}
```

Factorial

```
#include <stdio.h>

int main()
{
```

```
  int c, n, fact = 1;

  printf("Enter a number to calculate it's factorial\n");
  scanf("%d", &n);

  for (c = 1; c <= n; c++)
    fact = fact * c;

  printf("Factorial of %d = %d\n", n, fact);

  return 0;
}
```

**Prime number program in C**

```
#include<stdio.h>

int main()
{
   int n, i = 3, count, c;
  printf("Enter the number of prime numbers required\n");
   scanf("%d",&n);

   if ( n >= 1 )
   {
      printf("First %d prime numbers are :\n",n);
      printf("2\n");
   }

   for ( count = 2 ; count <= n ;  )
   {
      for ( c = 2 ; c <= i - 1 ; c++ )
      {
         if ( i%c == 0 )
            break;
      }
      if ( c == i )
      {
         printf("%d\n",i);
         count++;
      }
      i++;
   }
   return 0;
}
```

**Program to calculate GCD And LCM**

```
#include <stdio.h>

int main() {
  int a, b, x, y, t, gcd, lcm;
```

```
  printf("Enter two integers\n");
  scanf("%d%d", &x, &y);
  a = x;
  b = y;
  while (b != 0) {
    t = b;
    b = a % b;
    a = t;
  }
  gcd = a;
  lcm = (x*y)/gcd;
  printf("Greatest common divisor of %d and %d = %d\n", x, y, gcd);
  printf("Least common multiple of %d and %d = %d\n", x, y, lcm);
  return 0;
}
```

**Factors of a number**

```
#include <stdio.h>
int main()
{
    int number,i;

    printf("Enter a positive integer: ");
    scanf("%d",&number);
    printf("Factors of %d are: ", number);

    for(i=1; i <= number; ++i)
    {
        if (number%i == 0)
        {
            printf("%d ",i);
        }
    }

    return 0;
}
```

**Sine Series**

```
# include <stdio.h>
# include <conio.h>
# include <math.h>
void main()
{
 int i, n ;
 float x, val, sum, t ;
 clrscr() ;
 printf("Enter the value for x : ") ;
 scanf("%f", &x) ;
 printf("\nEnter the value for n : ") ;
```

```
 scanf("%d", &n) ;
 val = x ;
 x = x * 3.14159 / 180 ;
 t = x ;
 sum = x ;
 for(i = 1 ; i < n + 1 ; i++)
 {
  t = (t * pow((double) (-1), (double) (2 * i - 1)) *
   x * x) / (2 * i * (2 * i + 1)) ;
  sum = sum + t ;
 }
 printf("\nSine value of %f is : %8.4f", val, sum) ;
 getch() ;
}
```

Output of above program is

Enter the value for x : 30
Enter the value for n : 20
Sine value of 30.000000 is :   0.5000

**Cosine Series**

```
# include <stdio.h>
# include <conio.h>
# include <math.h>
void main()
{
 int i, n ;
 float x, val, sum = 1, t = 1 ;
 clrscr() ;
 printf("Enter the value for x : ") ;
 scanf("%f", &x) ;
 printf("\nEnter the value for n : ") ;
 scanf("%d", &n) ;
 val = x ;
 x = x * 3.14159 / 180 ;
 for(i = 1 ; i < n + 1 ; i++)
 {
  t = t * pow((double) (-1), (double) (2 * i - 1)) *
   x * x / (2 * i * (2 * i - 1)) ;
  sum = sum + t ;
 }
 printf("\nCosine value of %f is : %8.4f", val, sum) ;
 getch() ;
}
```

Output of above program is :
Enter the value for x : 60
Enter the value for n : 20
Cosine value of 60.000000 is :   0.5000

# UNIT-VI

## Introduction to Functions

Functions are the C building blocks where every program activity occurs. It is a self contained program segment that carries out some specific, well-defined task. Every C program must have a function. One of the function must be main( ).

C functions can be classified into two categories.

- Library functions — Predefined in the standard library of C. Need is just to include the library.
- User defined functions: — It has to be-developed by the user at the time of program writing.

### *Need of user Defined Functions*

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. This approach clearly results in a number of advantages.

- Length of program can be reduced by using function.
- Reusability of function increases.
- It is easy to use.
- Debugging is more suitable (easier) for programs.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.

### *Function Declaration and Prototypes*

Before defining the function, it is appropriate to declare the function along with its prototype.
In function prototype, the return value of function, type, and number of arguments are specified. The declaration of all functions statement should be first statements in main( ) or we can also declare globally for accessing all function within program.
The general form of function declaration is

**<return_type> <function_name> ( [<argument_list>] );**

Function prototypes are desirable because they facilitate error checking between calls to a function and corresponding function definition. They also help the compiler to perform automatic type conversions on function parameters.
When a function is called, actual arguments are automatically converted to the types in function definition using normal rules of assignment.

## *Function Definition*

The function Definition is, the task assigned to the function, that user declare. The general form of a function definition is:

```
return_type function_name (declarations of formal argument list)
    {

        local variable declarations;
        executable statement 1;
        executable statement 2;
        ---------------
        -----------
        -----------------
        executable statement n;
        return (expression);
    }
```

Where return_type represents the data type of the value which is returned. The type specification can be omitted if the function returns an integer or a character.

The formal argument list is a list of variables separated by commas that receive the values from main program when function is called.

The last statement in the body of function is return (expression). It is used to return the computed result, if any, to the calling program.

## *Calling a Function*

A function can be called by specifying its name followed by a list of arguments enclosed in parentheses and separated by commas. If a function call does not require any arguments, an empty pair of parenthesis must follow the function name.

**Example :**

```
#include <stdio.h>
#include <conio.h>
void add()
{
   int a, b, c;
   clrscr();
   printf("\n Enter Any 2 Numbers : ");
```

```
  scanf("%d %d",&a,&b);
  c = a + b;
  printf("\n Addition is : %d",c);
}
void main()
{
  Clrscr()
  void add(); // calling  add( ) function
  add();
  getch();
}
```

Output :

```
 Enter Any 2 Numbers : 23 6
 Addition is : 29
```

## ***The Return Statement:***

information is returned from the function to the calling portion of the program via return statement. Its uses control to be returned to the point from where the function was accessed. The return statement can take one of the following forms:

**return;**

**or**

**return (expression);**

In the return (expression); statement, the value of the expression is returned to the calling of the program.
A function can have multiple return statements, each containing different expression.

**Example:**

```
/*  Program to convert lowercase character to uppercase  */
# include <stdio.h>
main( )
{
char lower, upper;
char Lower_to_upper (char Lower);
printf  (''\n  Enter  the  lowercase  character:");
scanf ("%c", & Lower):
upper = Lower_to_upper (lower);
printf ("\n The upper case Equivalent is % c", upper);
```

```
}

char lower_to_upper (char ch)
{
char c2;
c2=  (c1  >  =  'a'  &&  c1  <  =  'z')  ?  (ch-32)  :  c1;
return (c2);
}
```

# Passing Parameters to Functions

## Call by value

Call by value means sending the values of the arguments to functions. When a Single value is passed to a function via-an actual argument, the value of the actual argument is copied into the function in formal argument, and no matter what the function does with that value, the value stored in actual argument remains unchanged.

This procedure to pass the value of an argument to a function is known as passing by value or call by value.

**Example:**

```
/* A simple C program containing a function that alters the value of its argument */
#include            <stdio.h>
void modify (int) ;
main( )
 {
      int N = 2;
      printf("\nN = %d (from main, before calling the function)",N);
      modify(N);  // sending actual value of N ( Call by value )
      printf("\nN = %d (from main, after calling the function)",N);
 }
void modify (int N)
{
        N * = 3;
        printf("\nN = %d (from the function, after being modified)",N);
}
```

**Output:** N= 2 (from main, before calling the function)

N = 6 (from the function, after being modified)

N = 2 (from main, after calling the function)

The original value of N (that is, =2) is displayed when main is executed. This value is then passed to the function modify, where it is multiplied by three and the new value of the formal argument that is displayed within the function. Finally, the value of N within main ( ) (the actual argument) is again displayed, after control is transferred back to function main from function modify.

These results show that N is not altered within main; even though the corresponding value of a is changed within modify.

## Call by Reference:

Call by reference means sending the addresses of the arguments to called function. In this method the addresses of actual arguments in the calling function are copied into formal arguments of the called functions. Thus using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. Using a call by reference it is possible to make a function return more than one value at a time.

**Example:**

```
/* A simple C program containing a function that alters the value of actual argument */

#include <stdio.h>
void    modify    (int    *)    ;
void main( )
 {
      int N = 2;
      printf("\nN = %d (from main, before calling the function)",N);
      modify(&N); // sending address of N ( Call by Reference )
      printf("\nN = %d (from main, after calling the function)",N);
 }
void  modify  (int  *N)    //  pointer  variable  to  receive  address  of  N
{
        *N= *N + 10;
        printf("\nN = %d (from the function, after being modified)",*N);
}
```

**Output:** N= 2 (from main, before calling the function)

N = 12 (from the function, after being modified)

N = 12 (from main, after calling the function)

## Passing Array to a function :

Array elements can be passed to a function by calling the function by value, or by reference. In call by value, we pass values of array elements to the function while in call by reference we pass the name of array, without any subscript, and the size of array elements to the function. Because array name is constant pointer that hold base address of first element (array[0]).

**Example :**

**/* passing Array by value * /**

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

void display(int); // function prototype

void main( )
{
  int array[5]={2,4,6,8,9};
  int i;
  clrscr();
  for(i=0;i<5;i++)
   {
    display(array[i]); // passing each array's  element one by one
   }
 getch();
}

void display(int n) //receive element in n
{
 printf("%d\t",n);
}
```

**Example :**

**/* passing Array by Reference * /**

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

```
void display( int[ ] ); // function prototype

void main( )
{
  int array[5]={2,4,6,8,9};
  int i;
  clrscr();
  display(array);  // passing base address of array
  getch();
}

void display(int n[] ) //receive element in array n
{
     int i;
     for(i=0;i<5;i++)
         {
           printf("%d\t",n);
         }
}
```

## Functions Returning values

The result of the function can be returned to the calling program. This defines the type-specifier in the function header. By default in C each function has a return type of "int". A function can return only one value. Arguments- are passed by reference to overcome this limitation.

If you want the function to return a value of any other data type than the default, you need to mention it explicitly in the function header (prototype).

The following example illustrates this

**Example :**

**/* finding largest value in an array * /**

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

int  large( int[ ] ); // function prototype

void main( )
```

```
{
  int array[5]={2,4,6,8,9};
  int i,max;
  clrscr();
  max=large(array);  //receive largest value in max returned by large( ) function
  printf("Largest value =%d", max);
  getch();
}

int large(int n[] ) //receive element in array n
{
     int i,m=n[0];
     for(i=1;i<5;i++)
         {
          If( m <n[ i ])
              m=n[ i ];
         }
         return m; //return value of m
}
```

## Command-line Arguments

Every program must have a function main( ). Till now, we know that main( ) function takes no arguments. But the empty parenthesis in the main( ) may contain special arguments that allow parameters to be passed to the main( ) from operating system.

Two arguments are passed to main( ) function

- **argc** : It is a integer type arguments that indicates the number of parameters passed to main( ) function. It automatically count the total number of arguments passed to function main().
- **argv** : It is a String (char * [ ] ) type arguments. Each String in this array will represent a parameter passed to function main().

**Example :**

```
void main( int argc , char *argv[ ] )
{
 int i;
 clrscr( );
 printf("Your Total argument Received in function main( ): %d\n", argc);
 printf("\nYour Given Arguments are :\n");
  for(i=1 ; i < argc ; i++)
  {
     printf("%s\n",argv[i]);
  }
```

```
getch();
}
```

Where argv and argc may have different name also. In the for loop we have print elements of String type array, named argv from index 1 to argc -1 because at index 0 array store executable file name <filename.exe> as argument.

**How to execute** :

**Step 1:** Create source file (filname.c)

**Step 2:** Press function key **F9** to make executable file

OR

Go to **Compile** menu and select **Make** option

**Step 3:** Go to **File** Menu and Select **Dos shell** option

**Step 4:** Type filename and specify arguments **i.e. filename arg1 arg2 arg3 arg4……………argN** and press Enter Key.

**Step 5:** See Output and type exit command to return in program.

**Arguments can be given as:**

```
Turbo C++ IDE
Type EXIT to return to Turbo C++. . .Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

C:\TC\BIN>Commline SABAB 45 10000 Microtek_
```

**Output:**

```
Turbo C++ IDE
Total argument Received in function main(): 5

Your Given Arguments are:
SABAB
45
10000
Microtek
```

# Recursion in Function

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

In order to solve a problem recursively, two conditions must be satisfied:

- Problem must be written in recursive form.
- Problem statement must include a stopping condition.

## *Recursive Function Definition*

Recursive function is a function that contains a call to itself. C supports creating recursive function with ease and efficient.

Recursive function must have at least one exit condition that can be satisfied. Otherwise, the recursive function will call itself repeatedly until the runtime stack overflows.

Recursive function allows you to divide your complex problem into identical single simple cases which can handle easily. This is also a well-known computer programming technique.

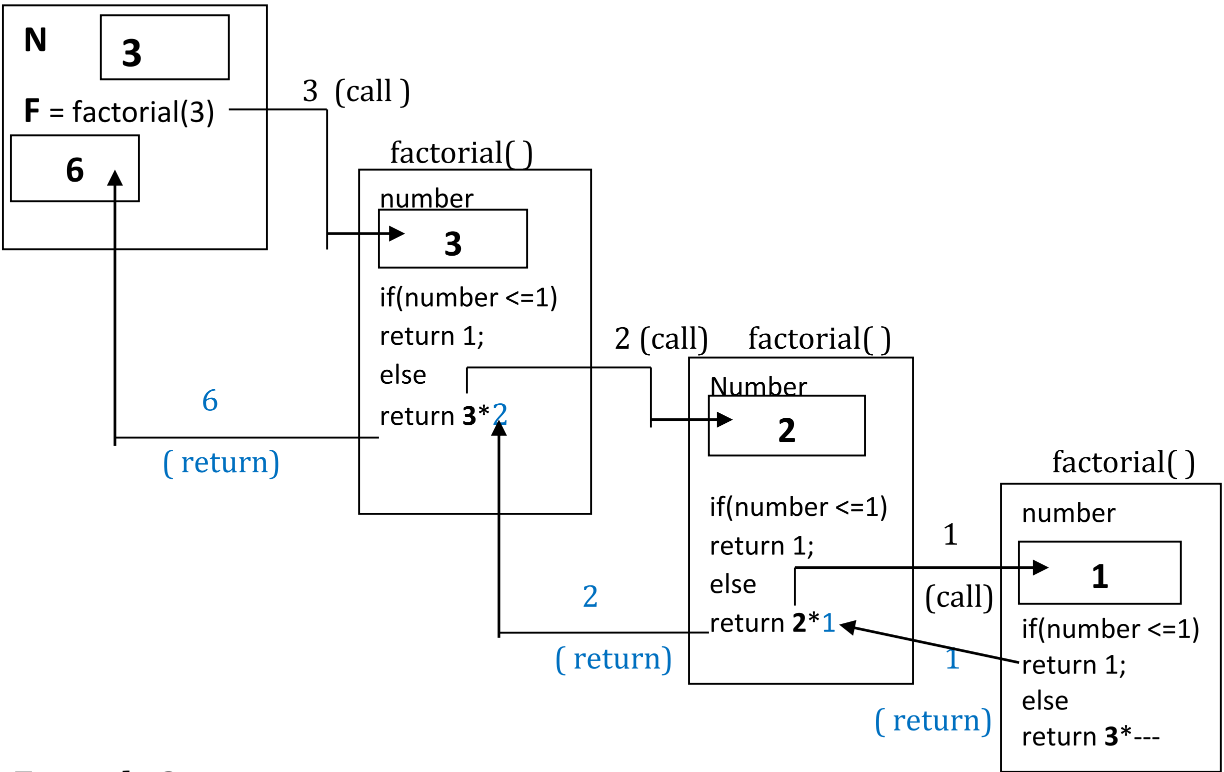
**Example 1:**

```
# include<stdio.h>
long unsigned int factorial( long unsigned int number)
{
 if(number <= 1)
        return 1;
else
        return number * factorial(number - 1);
}

void main()
{
        long unsigned int N,F;
        clrscr( );
        printf("Enter any Number for Calculate Factorial : ");
        scanf("%lu",&N);
        F=factorial(N);
        printf("\nFactorial of %lu is : %lu",N,F);
        getch( );
}
```

**Output:**

```
Enter any Number for Calculate Factorial : 5

Factorial of 5 is : 120
```

***Process diagram of program :***




**Example 2:** /* display numbers from 1 to 10 using recursive function */

```
# include<stdio.h>
void  display(int n)
{
 if(number > 10)
            return ;
else
     printf("%d\n",n);
  display(n+1);
}

void main()
{
            clrscr( );
            printf("Numbers from 1 to 10 \n");
            display(1); //1 is a starting number
            getch( );
}
```

# Storage Classes

There are two different ways to characterize variables.

- **by data types**
- **by storage class**

Data types refer to the type of information while storage class refers to the lifetime of a variable and its scope within the program.

A variable in C can have anyone of the four storage classes.

- **Automatic storage class**
- **Extern storage class**
- **Static storage class**
- **Register storage class**

A variable storage class tells us

1) Where the variable would be stored.
2) What will be the initial value of the variable, if the initial value is not specifically assigned (i.e. the default initial value).
3) What is the scope of the variable, i.e., in which functions the value of the variable would be available.
4) What is the life of the variables; i.e., how long would the variable exist.

## ***Automatic storage class***

These variables comes into existence whenever and wherever the variable is declared. These variables are also called as local variables, because these are available local to a function. The storage is in the memory and it has a default value, which is a garbage value. It is local to the block in which it has been declared and it life is till the control remains within the block in which the variable is defined. The key word used is '**auto**'.

By default..a variable declared inside a function with storage class specification is an **automatic.**

**Declaration:**

```
int N;
or
auto int N;
```

**Example :**

```
main()
{
auto int i=10;
printf("%d",i);
}
```

**Output:** 10

## *Extern storage class*

The variable has a storage in the memory. Default initial value is zero. The scope of the variable is global. It is present as long as the program execution comes to an end. The keyword used is '**extern**'. Default value of extern variable is zero.

**Declaration:** extern int N;

```
int i=10; // global variable
main()
{
  int i=2;
  printf("%d",i);
  display();
}
display()
{
  printf("\n%d",i);
}
```

**Output:**

2
10

```
main()
{
  int i=2;
  printf("%d",i);
  display();
}
display()
{
  extern int i;
  printf("\n%d",i);
}
int i=10; //global variable
```

**Output:**

2
10

In the following example, Variable " i " is a global variable. If the global variable declared outside (before function definition), there is no need to use extern declaration in function that use global variable. Whereas, if global variable declared outside (after function definition), it has to be extern declaration within function that use global variable.

## *Static storage class*

The storage is in the memory and default initial value is zero. It is local to the block in which it has been defined. The value of the variable persists between different function calls. The value will not disappear once the function in which it has been declared becomes inactive. It is unavailable only when you come out the program. The key word used is '**static**'.

**Declaration :** static int N;

**Example :**

```
void value()
{
  static int a=5;
  a=a+2;
  printf("\t%d",a);
}
```

```
void main(){
  value();
  value ();
  value();
  getch();
}
```

The output of the program is not 7 7 7

but it is **7 9 11**

## ***Register storage class***

The storage of this type of variables is in the CPU registers. It has a garbage value initially. The scope of the variable is it is local to the block in which the variable is defined. Its life is till the control remains in the block in which it is defined. A value stored in a CPU register cal always be accessed faster than the one which is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as register. A good example of frequently used variables is loop counters. The key word used is '**register**'.

**Declaration :**

```
register int N;
```

**Example :**

```
main()
{
register int i=10;
printf("%d",i);
}
```

**Output:** 10

# Scope and Lifetime of Declaration

| Storage Class | Where Declared | Visibility (Active) | Lifetime (Alive) |
|---|---|---|---|
| None | Before all functions in a file (may be initialized) | Entire file plus other files where variable is declared with extern | Entire Program (Global) |
| extern | Before all functions in a file (cannot be initialized) | Entire file plus other files where variable is. declared with extern and the files where variables originally declared as global | Global |
| static | Before all functions in a file | Only in that file | Global |
| auto | Inside a function or a block | Only in that function or block | Unit end of function or block |
| register | Inside a function or block | Only in that function or block | Unit end of function or block |
| static | Inside a function | Only in that function | Global |