

## Unit-1

### Algorithm and Flowchart

#### Algorithm:

Algorithm is a finite set of instructions that, if followed to accomplish a particular task. It can be defined as a sequence of instructions that when executed in the specified sequence, desired results are obtained. The same problem can be solved with different methods. So, to solve problem different algorithms, may be accomplished. Algorithm may vary in time, space utilized. User writes algorithm in his / her own language.

#### **Characteristics / Features of Algorithm:**

Algorithm should satisfy the following criteria:

1. *Input:* Zero or more quantities are externally supplied.
2. *Output:* At least one quantity is produced.
3. *Definiteness:* Each instruction is clear and unambiguous. Ex Add B or C to A
4. *Finiteness:* Algorithm should terminate after finite number of steps when traced in all cases e.g. Go on adding elements to an array
5. *Effectiveness:* Every instruction must be basic i.e., it can be carried out, by a person using pencil and paper.
6. *Independent of Programming:* Algorithm must also be general to deal with any situation.

#### **Advantages and Disadvantages of algorithm. (Pros and Cons / Merits and Demerits)**

##### **Advantages of Algorithms:**

1. It provides the core solution to a given problem. The solution can be implemented on a computer system using any programming language of user's choice.
2. It facilitates program development by acting as a design document or a blue print of a given problem solution.
3. It ensures easy comprehension of a problem solution as compared to an equivalent computer program.

# CPPM

4. It eases identification and removal of logical errors in a program.
5. It facilitates algorithm analysis to find out the most efficient solution to a given problem.

## **Disadvantages of Algorithms:**

1. In large algorithms the flow of program control becomes difficult to track.
2. Algorithms lack visual representation of programming constructs like flowcharts; thus understanding the logic becomes relatively difficult
3. There are no standard methods for writing algorithms, hence it can't be standardized.
4. It is time consuming task.

Algorithms can be categorised in three structures:

1. Simple Algorithm : Sequential steps are performed
2. Jumping and Branching Algorithm: Steps are executed based on certain criteria.
3. Looping Algorithms: As required in the problem certain steps can be repeated to solve a problem.

Depending upon the problems stated user can develop algorithm in any of the above structure.

### **Example1 : Add two numbers.**

Step 1: Start

Step 2: Read 2 numbers as A and B

Step 3: Add numbers A and B and store result in C

Step 4 : Display C

Step 5: Stop

### **Example2: Average of 3 numbers.**

1. Start

2. Read the numbers a , b , c

3. Compute the sum and divide by 3

4. Store the result in variable d

5. Print value of d

6. End

### **Example3: Average of n inputted numbers.**

1. Start

2. Read the number n

3. Initialize i to zero

4. Initialize sum to zero

5. If i is greater than n
6. Read a
7. Add a to sum
8. Go to step 5
9. Divide sum by n & store the result in avg
10. Print value of avg
11. End

## **Flowchart:**

A **flowchart** can be defined as a visual/pictorial representation of the sequence of steps for solving a problem. A flowchart is a set of symbols that indicate various operations in the program. For every process, there is a corresponding symbol in the flowchart. Once an algorithm is written, its pictorial representation can be done using flowchart symbols.

In other words, a pictorial representation of a textual algorithm is done using a flowchart.

- The first *flowchart* is made by John Von Neumann in 1945.
- It is a symbolic diagram of operations sequence, dataflow, control flow and processing logic in information processing.
- The symbols used are simple and easy to learn.
- It is a very helpful tool for programmers and beginners.

## **Purpose of a Flowchart:**

- Provides Communication.
- Provides an Overview.
- Shows all elements and their relationships.
- Quick method of showing Program flow.
- Checks Program logic.
- Facilitates Coding.
- Provides Program revision.
- Provides Program documentation.

## **Advantages/Merits/Pros/Positive Points of a Flowchart :**

1. Flowchart is an important aid in the development of an algorithm itself.
2. Easier to Understand than a Program itself.
3. Independent of any particular programming language.
4. Proper documentation.
5. Proper debugging.
6. Easy and Clear presentation.

## Disadvantages/Demerits/Cons/Negative points/Limitations/Hindrances/Problems of a Flowchart:

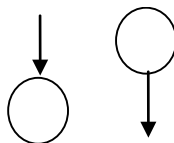
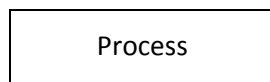
1. Complex logic.
2. Drawing is time consuming.
3. Difficult to draw and remember.
4. Technical detail.

## Flow charting Rules:

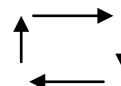
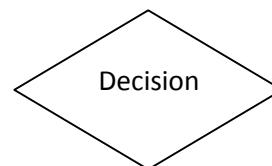
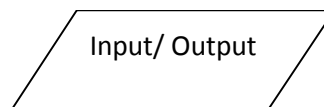
1. Maintain consistent level of details.
2. Chart only important and relevant instructions.
3. Use common and easy to understand statements
4. Be consistent in using .

## Symbols used in a Flowchart

The American National Standard Institute (ANSI) has standardized the flow chart symbols. Some of the common symbols used in flowcharts are shown below:



Connectors



Flow-Lines

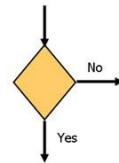
**Terminal:** The terminal symbol indicates the beginning (start) and (stop). In a program logic flow it also indicates halt (pause). It is the first and last symbol of the flowchart. A pause is normally used in a program logic under some error condition.

**Input/Output:** These symbols denote any function of an input/output nature in a program. All the program instructions to input/output data from any type of input/output device are indicated with input/output symbols in a flowchart. Instructions to Input/output data from a storage device are indicated with input/output symbols.

# CPPM

**Processing:** A processing symbol represents arithmetic and data movement instructions. All arithmetic process like adding, subtracting, multiplying and dividing are indicated by processing symbol in a flowchart. The logical processes of moving data from one location of the main memory to another are also denoted by this symbol.

**Decision:** The decision symbol indicates a decision point. i.e. a point at which a branch to one or two or more alternatives points is possible. During the execution appropriate path is followed depending upon the result of the decision.



**Flow-lines:** Flow lines with arrow heads indicate the flow of operation. That is exact sequence in which the instructions have to be executed. The normal flow of a flow chart is from top to bottom and left to right. The crossing of flow-lines should be avoided.

**Connectors:** When the flow chart becomes complex and big which causes the flow chart to spread over more than one pages; connectors are used. The connectors should be labelled properly to avoid confusion. The connectors can be used as entry to and exit from connector.

## Structured Programming

Structure programming is one of the approaches followed to develop program. In this approach the problem is divided into small modules or paths. These modules are small entities which perform specific task. Such program modules have one entry point and one exit point.

### **Advantages:**

- Easy to read and understand
- Easy to code
- Less time consuming
- Easy to maintain
- More reliable

## **Concepts of Compiler, Interpreter, Editor, Debugging & Testing**

The code written in High Level Language uses English like statements. Machine (Computer) recognizes machine language. Hence it is required to translate the code written in High Level Programming language to machine level language. i.e. Translators are required to translate HLL to MLL.

**Translator:** A program written in high-level language is called as source code. To convert the source code into machine code, translators are needed.

A translator takes a program written in source language as input and converts it into a program in target language as output.

# CPPM

It also detects and reports the error during translation.

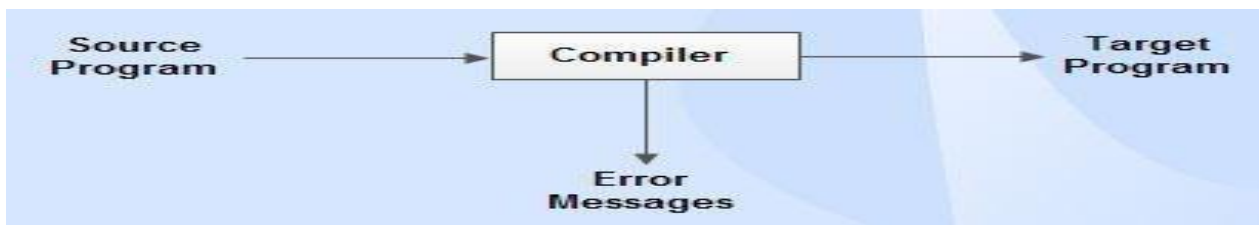
## Roles of translator are:

- Translating the high-level language program input into an equivalent machine language program.
- Providing diagnostic messages wherever the programmer violates specification of the high-level language program.

## Different type of translators

The different types of translator are as follows:

**Compiler:** Compiler is a translator which is used to convert programs in highlevel language to low-level language. It translates the entire program and also reports the errors in source program encountered during the translation.



## Advantages of using a compiler

- Source code is not included, therefore compiled code is more secure than interpreted code
- Tends to produce faster code than interpreting source code
- Produces an executable file, and therefore the program can be run without need of the source code

## Disadvantages of using a compiler

- Object code needs to be produced before a final executable file, this can be a slow process
- The source code must be 100% correct for the executable file to be produced

**Interpreter:** Interpreter is a translator which is used to convert programs in high-level language to low-level language. Interpreter translates line by line and reports the error once it encountered during the translation process.

It directly executes the operations specified in the source program when the input is given by the user. It gives better error diagnostics than a compiler.



# CPPM

## Advantages of using an Interpreter

- Easier to debug(check errors) than a compiler
- Easier to create multi-platform code, as each different platform would have an interpreter to run the same code
- Useful for prototyping software and testing basic program logic

## Disadvantages of using an Interpreter

- Source code is required for the program to be executed, and this source code can be read making it insecure
- Interpreters are generally slower than compiled programs due to the perline translation method

## Differences between compiler and interpreter

Sr. No	Compiler	Interpreter
1	Performs the translation of a program as a whole.	Performs statement by statement translation.
2	Execution is faster.	Execution is slower.
3	Requires more mem O56ry as linking is needed for the generated intermediate object code.	Memory usage is efficient as no intermediate object code is generated.
4	Debugging is hard as the error messages are generated after scanning the entire program only.	It stops translation when the first error is met. Hence, debugging is easy.
5	Programming languages like C, C++ uses compilers.	Programming languages like Python, BASIC, and Ruby uses interpreters.

## Assembler

Assembler is a translator which is used to translate the assembly language code into machine language code.



## Editor

# CPPM

A text editor is a computer program that lets a user enter, change, store, and usually print text (characters and numbers, each encoded by the computer and its input and output devices, arranged to have meaning to users or to other programs). Typically, a text editor provides an "empty" display screen (or "scrollable page") with a fixed-line length and visible line numbers. You can then fill the lines in with text, line by line. A special command line lets you move to a new page, scroll forward or backward, make global changes in the document, save the document, and perform other actions. After saving a document, you can then print it or display it. Before printing or displaying it, you may be able to format it for some specific output device or class of output device. Text editors can be used to enter program language source statements or to create documents such as technical manuals. E.g. C/C++ editors:

- Turbo c++ IDE
- DosBox
- gedit
- NetBeans □ Ecilpse

Etc...

## **Integrated Development Environments (IDE)**

The process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows us to easily manage large software programs, edit files in windows, and compile, link, run, and debug programs. On Mac OS X, CodeWarrior and Xcode are two IDEs that are used by many programmers. Under Windows, Microsoft Visual Studio is a good example of a popular IDE. Kylix is a popular IDE for developing applications under Linux. Most IDEs also support program development in several different programming languages in addition to C, such as C# and C++

## **Program Debugging and Testing**

Testing and Debugging is task of finding and removing errors. While programming the lots of errors are generated.

### **Types of errors**

**Syntax Error:** The programming language has specific format which has to be followed while writing code. Any violation in rule results in syntax error.

The compiler can detect and isolate such errors. The compilation process ends if any error is found and list of errors are displayed.

e.g. `printf("Hello")` here the C statement is not terminated with `;"`. Hence `;"` missing error" is listed



# CPPM

**Run-Time errors:** Errors that are found at run time are called run time error. E.g. mismatch of data type, array reference out of range etc. These error are undetected while compilation. Such errors will result in erroneous output.

**Logical error:** These errors are generated because of error in program logic. These errors are not detected by compiler. These errors are generated due to poor understanding of the problem. *If(x==y) printf("x and y are not equal");*

These errors are identified by debugging the program (i.e. Checking the program code line by line).

**Latent error:** These are hidden error and appear in the code when particular set of data are used.

e.g.  $(a+b)/(p-q)$

here when  $p=q$  then divide by zero occurs.

## **Program Testing**

Testing is the process of reviewing and executing a program with the intent of detecting errors. The errors can be of any types. Testing should include all the steps to detect all the possible errors in the program. Testing process may include the following two stages.

1. Human testing  
It follows code inspection; the entire code is analyzed statement by statement.
2. Computer based testing  
It includes Compiler testing and run time testing.

Program testing can be done either at module level (unit level) or at program level.

## **Debugging**

Debugging is the process of finding and resolving of defects or problems within the program that prevent correct operation of computer software or a system.

Debugging the program has following steps.

- Add break points in the c program where the errors are suspected.
- Run the program in debug mode.
- Check the variables by checking the values stored in that variable.
- We may check program step by step (i.e. line by line ) or we can skip certain code by adding break points.

Generally debugging process is mostly helpful in finding logical errors. Different editors have different commands for debugging the code.

# CPPM

e.g. F7,F8 are the keys used for debugging in turbo c++

cc -g command , gdb is used for debugging in unix.

## C-Programming

### History

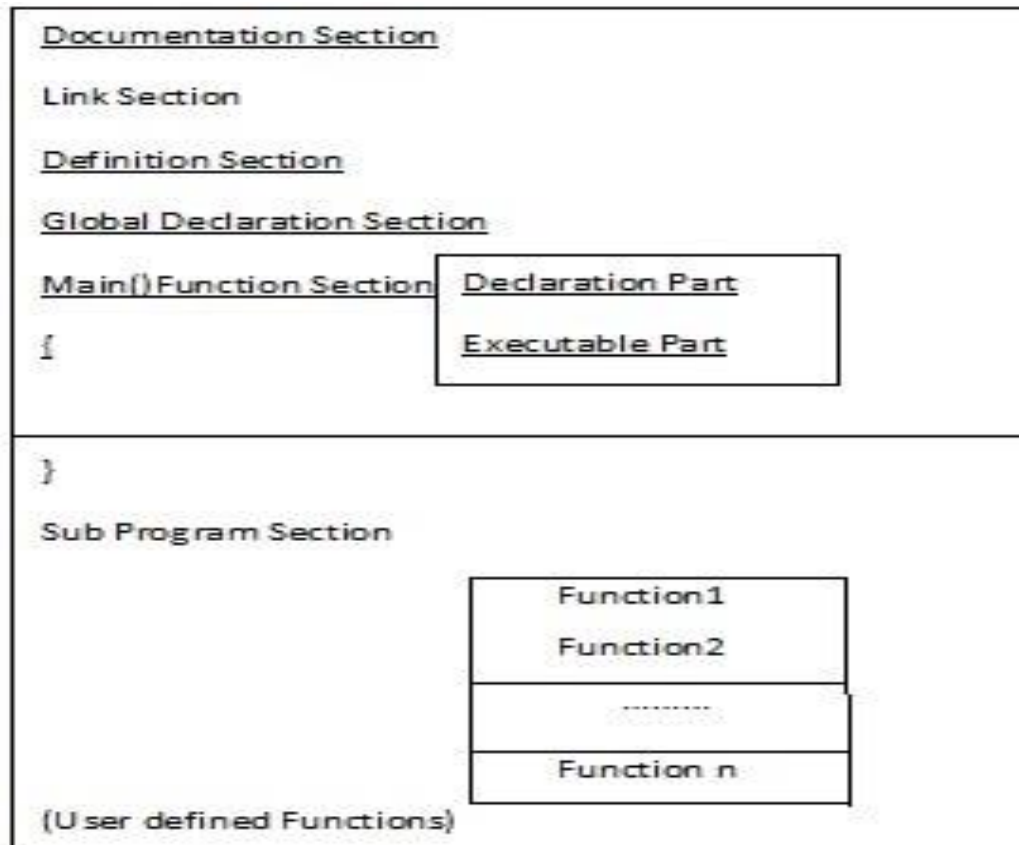
- Created by Dennis Ritchie at AT&T Labs in 1972
- Originally created to design and support the Unix operating system.
- There were only 27 keywords in the original version of C.
  - for, goto, if, else .....
- Easy to build a compiler for C.
- Many people have written C compilers
- C compilers are available for virtually every platform
- In 1983 the American National Standards Institute (ANSI) formed a committee to establish a standard definition. – Called ANSI Standard C.
- As opposed to K&R C (referring to the general “standards” that appeared in the first edition of Brian Kernighan and Ritchie’s influential book: The C Programming Language)

### Importance of C

- C is intended as a language for programmers □ There are 32 keywords in ANSI C.
- C has very rich set of built in functions.
- C is highly portable language, Program written on one machine can be run on another machine.
- C has ability to extend itself.
- C is powerful and efficient – You can nearly achieve the efficiency of assembly code.
- System calls and pointers allow you do most of the things that you can do with an assembly language.
- C is a structured language
- Code can be written and read much easier.
- C is standardized – Your ANSI C program should work with any ANSI C compiler.

### **Basic Structure of C program:**

# CPPM



**Documentation Sections:** This section consists of a set of comment lines giving the name of the program, and other details. In which the programme would like to user later. There are two types of comments in C programming:

Single Line comment(//):

// Program: To add two numbers

Multiline Comment (/\*.....\*/)

It is used when the user wants to comment multiple lines.

**Link section:** Link section provides instructions to the compiler to link functions from the system library.

Ex:- # include<stdio.h>

# include<conio.h>

**Definition section:** Definition section defines all symbolic constants.

Ex:- # define A 10.

**Global declaration section:** Some of the variables that are used in more than one function throughout the program are called global variables and declared outside of all the functions. This section declares all the user-defined functions.

Every C program must have one main ( ) function section. This contains two parts.

# CPPM

**i) Declaration part:** This part declares all the variables used in the executable part. Ex:- int a,b;

**ii) Executable part:** This part contains at least one statement. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. All the statements in the declaration and executable parts end with a semicolon (;).

**Sub program section:** This section contains all the user-defined functions, that are called in the main () function. User-defined functions generally places immediately after the main() function, although they may appear in any order.

This void means "main"  
Returns no value.

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

main function

Library /Header file/ Prototype

```
void main(void)
```

```
{
```

This void means "main" passes no argument.

```
printf("How are you!");
```

printf function is used to display the information, which is written inside its braces.

Body of the main function.

```
}
```

## Executing A "C" program

Executing C program involves a series of steps. The steps are:

1. Creating the program
2. Compiling the program
3. Linking the program with functions of C library
4. Executing the program

**Creating the program:** The program is created using any of the editor that support c programming. The program must be saved using ".C" extension.

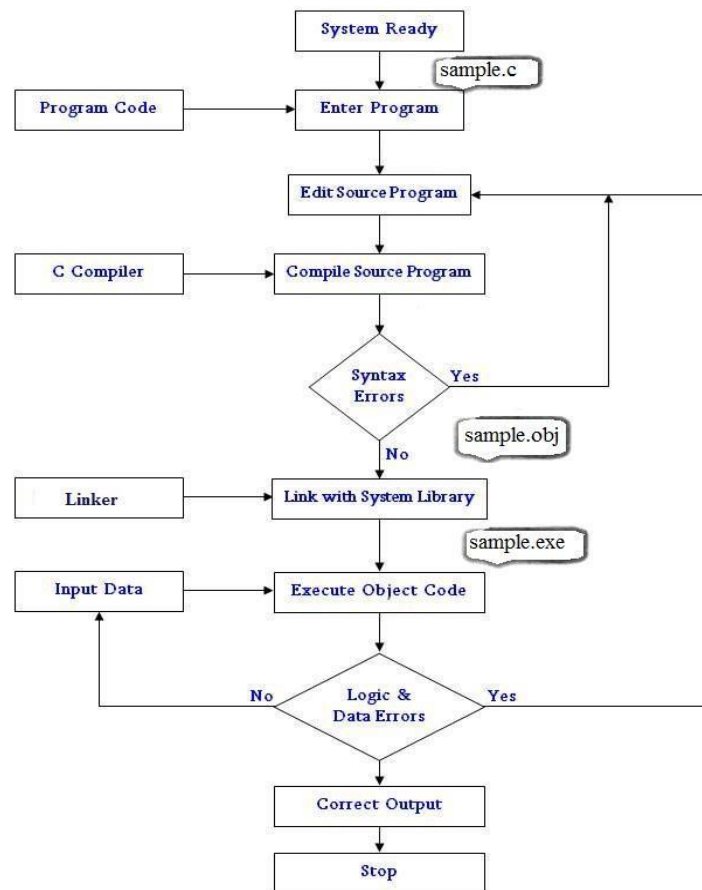
**Compiling C program:** The C-program (i.e. source code) is complied by the c compiler. The syntax error are listed if any the code has to be modified and complied again. If it is error free then object file is generated (.obj) it is passed for linking.

**Linking:** In this phase the c library functions and c program code is put together for further processing. Executable code is ready.

# CPPM

**Execution:** The executable code is executed. It may generate errors. It could be because of:

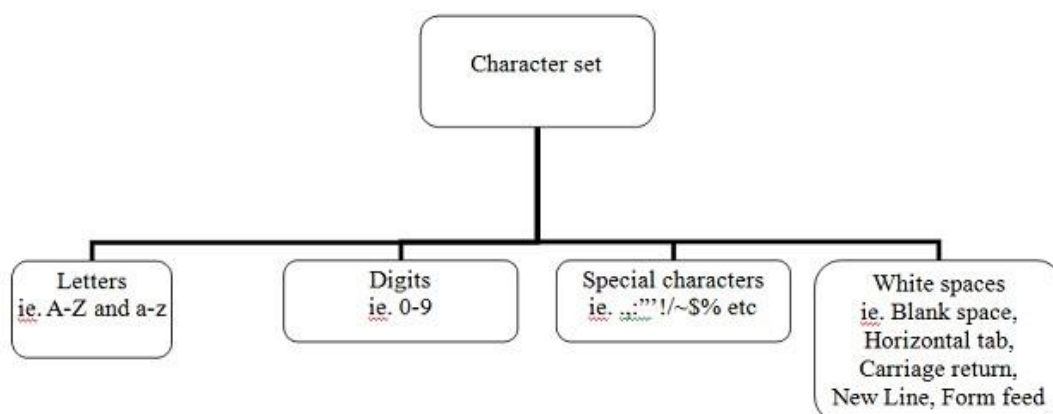
- Wrong logic: The code has to be modified and all the steps are repeated.
- Wrong Data Input: The input has to be given proper.



Process of compiling and running a C program.

## Character Set

Character set is a set of characters that denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers, white spaces and special symbols allowed in C. The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

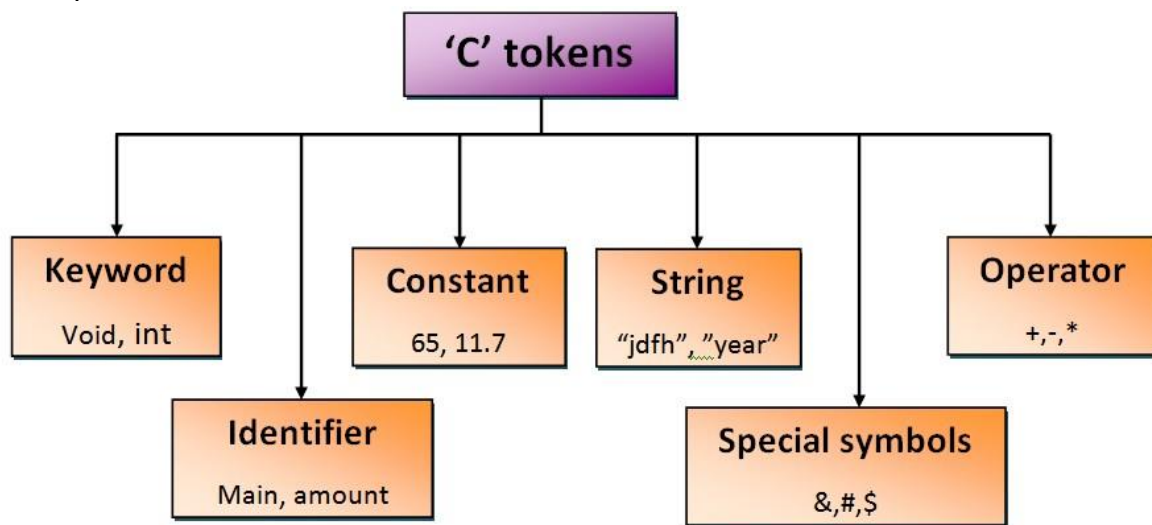


# CPPM

## C tokens

The smallest individual unit in a c program is known as a token or a lexical unit. C Tokens are the smallest building block or smallest unit of a C program. C tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators



## **Keyword**

There are certain words reserved for doing specific task, these words are known as reserved word or keywords. These words are predefined and always written in lower case or small letter. There are 32 keywords defined in ANSI C.

These keywords can't be used as a variable name as it assigned with fixed meaning.

### **Some examples are:**

int, short,	long,	for,	enum,
signed,	double,	union,	case,
unsigned,	break,	return,	goto,
default,	continue,	while, do,	struct,
volatile,	typedef,	extern,	char,
float,	static, do,	register,	auto,
			const etc.

## **Identifiers**

# CPPM

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc.

## Rules for naming identifiers are:

1. Name should only consists of alphabets (both upper and lower case), digits and underscore ( \_ ) sign.
2. Identifier names must be unique.
3. First characters should be alphabet or underscore
4. Name should not be a keyword.
5. Since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.
6. Identifiers are generally given in some meaningful name, such as value, net\_salary, age, data etc.
7. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters.
8. Must not contain white spaces.

## Constants and Variables – Needs & Definition

**Variables:** It is a data name that is used to store a data value. It can be changed during the execution of a program. A variable may take different values at different times during execution. Variables are memory locations (storage area) in C programming language. The primary purpose of variables is to store data in memory for later use.

**Rules:** Variable names may consist of letters, digits and under score( \_ ) character.

1. First char must be an alphabet or an '-'
2. Length of the variable cont exceed upto 8 characters, some C compilers can be recognized upto 31 characters.
3. No ", " and no white space, special symbols allowed.
4. Variables name should not be a keyword.
5. Both upper & lower case letters are used.

Ex:- mark,sum1,tot-value,delhi are valid

Prics\$, group one, char are invalid

18. How to declare and initialize a variable?

## Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

# CPPM

The declaration of variables must be done before they are used in the program.

**The syntax for declaring a variable is as follows:**

## **Syntax:**

**DataType V1, V1....Vn;**

V1,v2,...vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example , valid declarations are:

```
int count; int  
number, total;  
double ratio;  
short int count;  
long int amount;  
double deviation;  
unsigned n;  
char c;
```

## **Initialization of variable :**

Initialize a variable in c to assign it a starting value. Without this we can't get whatever happened to memory at that moment.

C does not initialize variables automatically. So if you do not initialize them properly, you can get unexpected results. Fortunately, C makes it easy to initialize variables when you declare them.

## **For Example :**

```
int    x=45;          int    month_lengths[]    =  
  
{23,34,43,56,32,12,24};
```

**Constants:** Constants are like a variable, except that their value never changes during execution once defined.

Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

- Constants are also called literals.
- Constants can be any of the data types.
- It is considered best practice to define constants using only upper-case names.

## **TYPES OF C CONSTANTS**

1. Integer constants
2. Real constants



# CPPM

## 3. Character constants

## 4. String constants

### Syntax:

```
const type constant_name;
```

### Example:

```
const int SIDE = 10;
```

**1. Integer constants:** An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal integer, octal integer and hexadecimal integer.

Examples of Integer Constant: 426 , +786 , -34 (decimal integers)

037, 0345, 0661 (octal integers)

0X2, 0X9F, 0X (hexadecimal integers)

### RULES OF CONSTRUCTING INTEGER CONSTANTS

- an integer constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- The allowable range for constants is -32768 to 32767

In fact the range of integer constants depends upon compiler. For ex. 435 +786 - 7000

**2. Real constants:** These quantities are represented by numbers containing fractional parts like 18.234. Such numbers are called real (or floating point) constants.

Examples of Real Constants: +325.34 426.0 -32.67 etc. The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. In exponential form of representation the Real Constant is represented in two parts. The first part present before 'e' is called Mantissa and the part following 'e' is called Exponent. For ex. .000342 can be written in Exponential form as 3.42e-4.

### RULES OF CONSTRUCTING REAL CONSTANTS

- A real constants must have at least one digit
- it must have a decimal point.
- it could be either positive or negative.
- default sign is positive. For ex. +325.34 426.0

In exponential form of representation, the real constants is represented in two parts. The part appearing before 'e' is called mantissa where as the part following 'e' is called exponent. Range of real constants expressed in exponential form is - 3.4e38 to 3.4e38. Ex. +3.2e-5

# CPPM

**3. Single Character constants:** Single character constant contains a single character enclosed within a pair of single quote marks.

For ex. 'A','5',';',',','\'

Note that the character constant '5' is not same as the number 5. The last constant is a blank space. Character constant has integer values known as ASCII values. For example, the statement

Printf("%d",a); would print the number 97, the ASCII value of the letter a. Similarly, the statement printf("%c",97); would output the letter 'a'

**4. String constants:** A string constant is a sequence of character enclosed in double quotes. the characters may be letters, numbers, special characters and blank space.

Examples are:

"HELLO!"

"1979"

"welcome"

"?.....!"

"5+3"

"X"

## RULES OF CONSTRUCTING CHARACTER CONSTANTS

- a. a character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
- b. The maximum length of character constant can be one character. Ex 'A'

**String:** In C programming, the one-dimensional array of characters are called strings, which is terminated by a null character '\0'. There are two ways to declare a string in C programming: (1) Through an array of characters. (2) Through pointers.

## **DATA TYPES**

Data type is the type of the data that are going to access within the program. C supports different data types. Each data type may have pre-defined memory requirement and storage representation. C supports 4 classes of data types.

1. Primary or (fundamental) data type(int, char, float, double)
2. User-defined data type (type def)
3. Derived data type (arrays, pointers, structures, unions)
4. Empty data type (void)- void type has no value.

1 byte = 8 bits (0's and 1's)

### **1. Primary or (fundamental) data type**

All C compilers support 4 fundamentals data types, they are;

**UCCC & SPBCBA & SDHGCBCA & IT College**

# CPPM

DATA TYPES	RANGE	Size	Control string
char	-128 to +127	1 byte	%c
int	-32768 to +32767	2 bytes	%d (or) %i
float	1.2E-38 to 3.4E+38	4 bytes	%f
double	2.3E-308 to 1.7E+308	8 bytes	%lf

## Integer types:

Integers are whole numbers with a range of values supported by a particular machine. Integers occupy one word of storage and since the word size of the machine vary. If we use 16 bit word length the size of an integer value is -32768 to +32767. In order to control over the range of numbers and storage space, C has 3 classes of integer storage, namely short, long, and unsigned.

# CPPM

DATA TYPES	RANGE	Size	Control string
<b>int (or) signed int</b>	<b>-32768 to +32767</b>	<b>2 bytes</b>	<b>%d (or) %i</b>
unsigned int	0 to 65535	2 bytes	%u
signed short int (or) short int	-128 to 127	1 byte	%d (or) %i
unsigned short int	0 to 255	1 byte	%d (or) %i
long int (or) signed long int	-2147483648 to +2147483647	4 bytes	%ld
unsigned long int	0 to 4294967295	4 bytes	%lu

## Character type:-

Single character can be defined as a character (char) type data. Characters are usually stored in 8 bits of internal storage. Two classes of char types are there. Signed char, unsigned char

DATA TYPES	RANGE	Size	Control string
<b>signed char (or) char</b>	<b>1 byte</b>	<b>-128 to +127</b>	<b>%c</b>
unsigned char	1 byte	0 to 255	%c

## Floating point types:

Type	Storage size	Value range	Precision	Control string
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places	%f
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places	%lf

# CPPM

long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places	%lf
-------------	---------	------------------------	-------------------	-----

## 2. Derived data type:

Derived data types are the data types that are derived from the primary data types.

These data types are declared using primary data types e.g. arrays, pointers, structures, unions

int a[10] --> is "a" array

## 3. User defined data types:

C –supports a feature known as "**type definition**" that allows users define an identifier that would represents an existing type.

Ex:- typedef data-type identifier;

Where data-type indicates the existing datatype

Identifier indicates the new name that is given to the data type.

Ex:- typedef int marks;

Marks m1, m2, m3;

typedef cont create a new data type ,it can rename the existing datatype. The main advantage of typedef is that we can create meaningful datatype names for increasing the readability of the program.

Another user-defined datatype is "**enumerated data type**(enum)"

Syntax: enum identifier{value1, value2,.....valuen};

Where identifier is user-defined datatype which is used to declare variables that can have one of the values enclosed within the braces. Value1 ,value2,.valuen all these are known as enumeration constants.

Ex:- enum identifier v1, v2,.....vn

V1=value3;

V2=value1;.....

Ex:- enum day {Monday,Tuesday..... sunday};

Enum day week-f,week-end Week-f

= Monday

enum day{Monday...Sunday}week-f, week-end;

## 4.Empty data type (void):

void type has no value. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function.

### Questions of Unit-1

1. What is void main()?
2. What is difference between compiler and interpreter?
3. Write a short note on algorithm.
4. Explain flowchart in detail.
5. Write a short note on structured programming.
6. Explain compiler and interpreter in detail.
7. Write a short note on basic structure of C program.
8. Write a short note on data type of C.
9. Write a short note on debugging.
10. Explain about types of error.
- 11.Explain Constant in detail.

## Unit 2.

### Expression & Operators

**Operators:** An operator is a symbol performs certain mathematical or logical manipulations. Operators are used in programs to manipulate data variables.

C operators can be classified into a number of categories, they are:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Unary Operators  
Increment and decrement operators
6. Conditional operators

### 1. Arithmetic Operators:

The arithmetic operators are Operator which are used to do arithmetic operations such as;	
+	Addition

# CPPM

-	Subtraction
*	Multiplication
/	Division
%	Modulo division

Here a and b are operands, assign values for a=14 and b=4 we have the following results  
 $a-b = 10$   $a+b = 18$   $a*b = 56$   $a/b = 3$ (coefficient)  
 $a\%b = 2$ (remainder)

## 2. Relational Operators:

Relational operators are used for comparing two quantities, and take certain decision. For example we may compare the age of two persons or the price of two items....these comparisons can be done with the help of relational operators. An expression containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false.

Ex:-  $13 < 34$  (true)  $23 > 35$ (false)

C supports 6 relational operators

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

Ex:-  $4.5 \leq 10$ (true)

$6.5 < -10$ (false)

$10 < 4+12$ (true)

When arithmetic expression are used on either side of a relational operator, the arithmetic expression will be evaluated first and then the results compared, that means arithmetic operators have a higher priority over relational operators.

## 3. Logical Operator:

C has 3 logical operators. The logical operators are used when we want to test more than one condition and make decisions.

Operator	Meaning
&&	Logical AND

# CPPM

	Logical OR
--	------------

The logical operators && and || are used when we test more than one condition and make decisions. Ex:-  $a > b$  &&  $x == 45$

This expression combines two or more relational expressions, is termed as a logical expression or a compound relational expression. The logical expression given above is true only if  $a > b$  is true and  $x == 10$  is true. if both of them are false the expression is false.

OP1	OP2	&&	
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

## 4. Assignment operator:

These operators are used to assign the result of an expression to a variable. The usual assignment operator is '='

$V \text{ op} = \text{exp};$

Where v is a variable, exp is an expression and op is a C binary arithmetic operator.

The operator op= is known as the shorthand assignment operator.

The assignment statement is  $V \text{ op} = \text{exp};$

Ex:-  $X = X + (Y + 1);$   $a * = a;$

$a = a * a;$

## 5. Unary Operator

- Increment (++)**
- Decrement operator(--)**
- Unary Plus (+)**
- Unary Minus(-)**

$++$  and  $--$  are increment and decrement operators in C. The operator  $++$  adds 1 to the operand, while  $--$  subtracts 1. both are unary operators.  $++m;$  or  $m++;$  is equal to  $m = m + 1 (m += 1;)$

$--m;$  or  $m--$  is equal to  $m = m - 1 (m -= 1;)$

We use the increment and decrement statements in for and while loops extensively.

Ex:-  $m = 5;$

$Y = ++m;$  the value of  $y = 6$  and  $m = 6.$

Suppose if we write the above statement as  $m = 5;$

$y = m++;$  the value of  $y = 5$  and  $m = 6.$

A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.



## 6. Conditional operator:

A ternary operator pair “?:” is available in C to construct conditional expressions of the form `exp ? exp : exp3`

Where `exp1`, `exp2` and `exp3` are expressions,

The operator “?:” works as follows: `exp1` is evaluated first. If it is non-zero (true), then the expression `exp 2` is evaluated and becomes the value of the expression. If `exp1` is false, `exp3` is evaluated and its value becomes the value of the expression.

Ex:- `a=10; b=45;`

`X = (a>b) ? a:b;`

o/p:- X value of b (45).

### **Expressions:**

An expression in C is some combination of constants, variables, operators and function calls.

Sample expressions are:

`a + b tan(angle)`

#### 2.2.1 Arithmetic expression

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. C can handle complex mathematical expressions.

e.g.

`a*b-c, (m+n)*(x+y), a*(b/c), 3*x*x 2*x+1, x/y+c`

#### 2.2.2 Boolean expression

Boolean expression is an expression that has relational and/or logical operators operating on Boolean variables. A Boolean expression evaluates to either true or false.

e.g.

`(!n >0), a<=b, a==b && b==c, a!=b || c<=d`

#### 2.3 Evaluation & Assignment of Expression

- Most expressions have a value based on their contents.
- A statement in C is just an expression terminated with a semicolon.

# CPPM

For example:

```
sum = x + y + z;  
printf("Go Buckeyes!");
```

The rules given below are used to evaluate an expression,

- 1) If an expression has parenthesis, sub expression within the parenthesis is evaluated first and arithmetic expression without parenthesis is evaluated first.
- 2) The operators high level precedence are evaluated first.
- 3) The operators at same precedence are evaluated from left to right or right to left depending on the associativity of operators.

Expressions are evaluated using an assignment statement of the form: Variable = expression;

Variable is any valid C variable name. when the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Ex:- x = a\*b-c;

## Operator Precedence:

Various relational operators have different priorities or precedence. If an arithmetic expression contains more operators then the execution will be performed according to their properties. The precedence is set for different operators in C.

Type of operator	Operators	Associativity
Unary operators	+, -, !, ++, --, type, , size of	Right to left
Arithmetic operators	*, /, %, +, -	Left to right
Bit-manipulation operators	<<, >>	Left to right
Relational operators	>, <, >=, <=, ==, !=	Left to right
Logical operators	&&,	Left to right
Conditional operators	?, :	Left to right

# CPPM

Assignment operators	=, +=, -=, *=, /=, %=	Right to left
----------------------	-----------------------	---------------

Important note:

- ☐ Precedence rules decide the order in which different operators are applied
- ☐ Associativity rule decides the order in which multiple occurrences of the same level operator are applied.

There are some operators which are given below with their mean. The higher the position of an operator is, higher is its priority.

Hierarchy Of Operations In	Type
!	Logical NOT
* / %	Arithmetic and modulus
+ -	Arithmetic
< > <= >=	Relational
== !=	Relational
&&	Logical AND
	Logical OR
=	Assignment

## ASSOCIATIVITY OF OPERATOR

When an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators.

Associativity can be of two types

—Left to Right

—Right to Left.

Left to Right associativity means that the left operand must be unambiguous. I.e. it must not be involved in evaluation of any other sub-expression. Similarly, in case of Right to Left associativity the right operand must be unambiguous. Let us understand this with an example. Consider the expression  $a = 3 / 2 * 5$  ;

# CPPM

Here there is a tie between operators of same priority, that is between / and \*. This tie is settled using the associativity of / and \*. But both enjoy Left to Right associativity.

While executing an arithmetic statement, which has two or more operators, we may have some problem as to how exactly does it get executed.

Priorit y	Operators	Description
1st	*, / , %	multiplication, division, modular division
2nd	+, -	addition, subtraction
3rd	=	assignment

**Type conversions:** converting a variable value or a constant value temporarily from one data type to other data type for the purpose of calculation is known as type conversion.

There are two types of conversions

1. Automatic type conversion (or) implicit

2. Casting a value (or) explicit

**1. Implicit type conversion:** C permits mixing of variables, constants of different types in an expression. In this higher data type can be converted into lower data type. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing its meaning. This automatic conversion is known as implicit type conversion.

- Float value can be converted into integral value by removing the fractional part.
- Double value can be converted into float value by rounding of the digits. □  
Long int can be converted into int value by removing higher order bits.

During the evaluation of expression, If the operands are of different types the lower type is automatically converted to the higher types before operation proceed. Hence, the result is of higher type.

The following diagram shows the type conversion process.

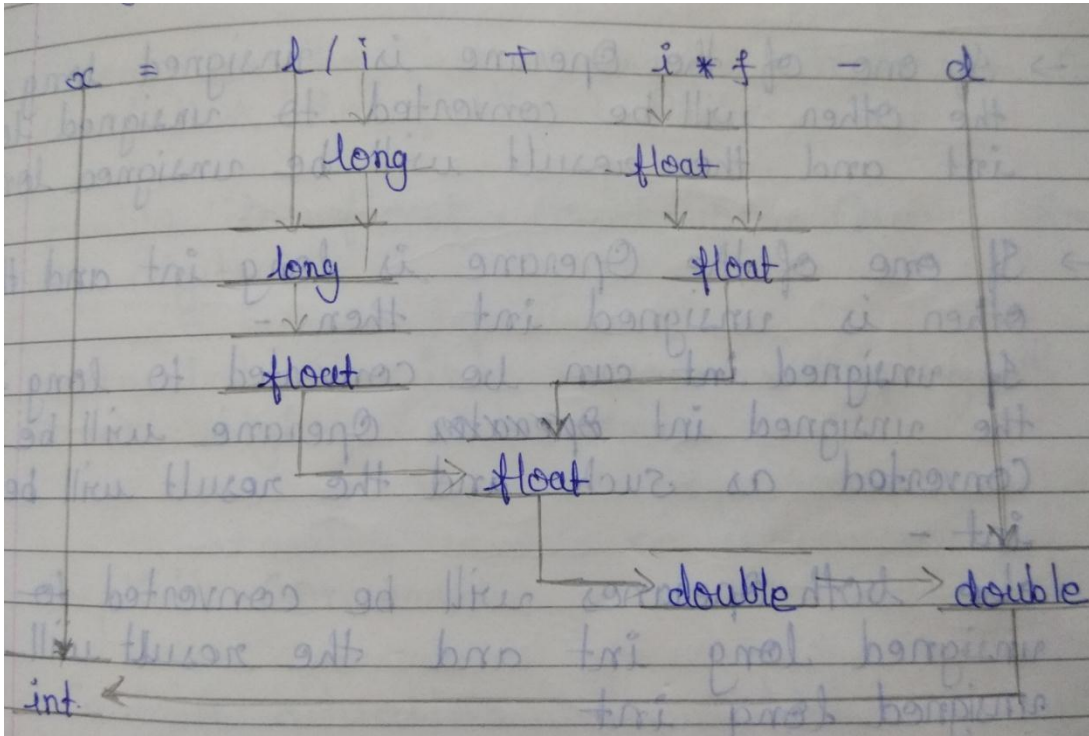
# CPPM

int i,x; float

f; double

d; long int

l;



The following rules are applied while evaluating an expression. All short int and character data type are automatically converted to the int.

If one of the operands is	Will be converted to	Result
long double	long double	long double
Double	double	Double
Float	float	Float
Long int	Long int	Long int
Unsigned int	Unsigned int	Unsigned int

If one of the operand is long int and the other is unsigned int then- if unsigned int can be converted to long int the unsigned int operand will be converted as such and the result will be long int-

Else both operands will be converted to unsigned long int and the result will be unsigned long int.

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

# CPPM

The following changes are performed during the final assignment:

- Float to the int conversion causes truncation of the fractional part
- Double to float conversion causes rounding of digits
- Long int to int causes dropping of the excess higher order bits

**2. Explicit:** In this type of conversion, the programmer can convert one data type to other data type explicitly.

Syntax: (datatype) (expression)

Expression can be a constant or a variable

Ex: `y = (int) (a+b)` `y= cos(double(x))`

`double a = 6.5` `double b = 6.5` `int result`

`= (int) (a) + (int) (b)` `result = 12` instead  
of 13.

`int a=10` `float(a)->10.00000`

## Unit -2 Questions

1. What is conditional operator? Explain in brief.
2. What is the use of relational operator?
3. What is type casting? Give proper example.
4. Explain `? :` operator?
5. What is the difference between post and pre increment operator? Explain with example.
6. Explain C operators in detail.

## Unit 3

### Input/ Output Statements & Built-in Functions

Formatted I/O statements (like scanf, printf)

Unformatted I/O statements (like getchar(), getch(), getche(), putchar())

Conversion Functions

#### **Managing input and output operations:**

Reading, processing and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data. We have two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements like `x=5`, `a=0` and so on. Another method is to use the input function `scanf`, which can read data from a keyboard. We have used both the methods in programs. For outputting results, we have used extensively the function `printf`, which sends results out to a terminal.

Input – Output functions:-

- ☐ The program takes some I/P- data, process it and gives the O/P.
- ☐ We have two methods for providing data to the program
  - i) Assigning the data to the variables in a program.
  - ii) By using I/P-O/P statements.

C language has 2 types of I/O statements; all these operations are carried out through function calls.

1. Unformatted I/O statements

2. Formatted I/O statements **Unformatted I/O statements:-**

# CPPM

**getchar( ):-** It reads single character from standard input device. This function don't require any arguments.

Syntax:- `char variable_name = getchar( );`

Ex:- `char x; x = getchar( );`

**putchar ( ):-** This function is used to display one character at a time on the standard output device.

Syntax:- `putchar(char_variable);`

Ex:- `char x; Putchar(x);`

`void main( )`

`{`

`Char ch;`

`Printf("enter a char");`

`Ch = getchar( );`

`Printf("enter char is");`

`Putchar(ch);`

`}` **getc() :-** This function is used to accept single character from the file.

Syntax: `char variable_name = getc();` Ex:- `char c; c =`

`getc();` **putc():-** This function is used to display single character.

Syntax:- `putc(char variable_name);`

Ex:- `charc;`

`Putc(c);`

These functions are used in file processing.

**gets( ):-** This function is used to read group of characters(string) from the standard I/P device.

Syntax:- `gets(character array variable);`

Ex:- `gets(s);`

**Puts( ):-** This function is used to display string to the standard O/P device.

Syntax:- `puts(char array variables);` Ex:-

`puts(s);`



# CPPM

```
void main()
{
    char s[10];
    Puts("enter name");
    gets(s);
    puts("print name");
    puts(s);
}
```

**getch():-** This function reads single character directly from the keyboard without displaying on the screen. This function is used at the end of the program for displaying the output (without pressing (Alt-F5).

Syntax: char variable\_name = getch();

Ex:- char c c = getch();

**getche():-** This function reads single character from the keyboard and echoes(displays) it to the current text window.

Syntax:- char variable\_name = getche();

Ex:- char c c = getche();

```
void main()
{
    char ch, c; printf("enter
char"); ch = getch();
    printf("%c", ch);
    printf("enter char");
    c = getche();
    printf("%c",c);
}
```

Character test function:- Function	Test
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character
isdigit(c)	Is c is a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c printable character?

# CPPM

ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a while space character?
isupper(c)	Is c an upper case letter?
tolower(c)	Convert ch to lower case
toupper(c)	Convert ch to upper case

```
Ex:- void
main()
{
    char a;
    printf("enter char");
    a = getchar();    if
(isupper(a))
    {
        x= tolower(a);
        putchar(x);
    }
    else
    {
        putchar(toupper(a));
    }
}
```

**O/P:-** enter char A a

**Formatted I/O Functions:** Formatted I/O refers to input and output that has been arranged in a particular format.

Formatted I/P functions ☐ scanf( ) , fscanf()

Formatted O/P functions---☐ printf() , fprintf()

**scanf( ) :-** scanf() function is used to read information from the standard I/P device.

Syntax:- scanf("controlstring", &variable\_name);

Ex:- int n;

Scanf("%d",n);

Control string represents the type of data that the user is going to accept. & gives the address of variable.(char-%c , int-%d , float - %f , double-%lf). Control string and the variables going to I/P should match with each other.

# CPPM

Simple format specification as follows

%w type – specified ex:- %4d , %6c.....

Here 'w' represents integer value specifies total number of columns.

Ex:- scanf("%5d",&a); a = 4377

	4	3	7	7
--	---	---	---	---

**Printf( ):** This function is used to output any combination of data. The outputs are produced in such a way that they are understandable and are in an easy to use form. It is necessary for the programmer to give clarity of the output produced by his program.

Syntax:- printf("control string", var1,var2.....);

Control string consists of 3 types of items

1. Chars that will be printed on the screen as they appear.
2. Format specifications that define the O/P format for display of each item.
3. Escape sequence chars such as \n , \t and \b.....

## O/P of integer number: -

Format specifications %wd

### Format

Printf("%d", 9876)

Printf("%6d", 9876)

Printf("%2d", 9876)

Printf("%-6d", 9876)

Printf("%06d", 9876)

### O/P

9	8	7	6
---	---	---	---

		9	8	7	6
--	--	---	---	---	---

9	8
---	---

9	8	7	6		
---	---	---	---	--	--

0	0	9	8	7	6
---	---	---	---	---	---

## O/P of real number: %w.pf

w---- Indicates minimum number of positions that are to be used for display of the value. p-----Indicates number of digits to be displayed after the decimal point.

# CPPM

**Format**      **y=98.7654**

**O/P**

Printf("%7.4f",y)

9	8	.	7	6	5	4
---	---	---	---	---	---	---

Printf("%7.2f",y)

		9	8	.	7	7
--	--	---	---	---	---	---

Printf("%-7.2f",y)

9	8	.	7	7		
---	---	---	---	---	--	--

Printf("%10.2e",y)

		9	.	8	8	e	+	0	1
--	--	---	---	---	---	---	---	---	---

## O/P of single characters and string:

%WC

%WS

**Format**

**O/P**

%s

R	a	j	u		R	a	j	e	s	h		R	a	n	i
---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---

%18s

	R	a	j	u		R	a	J	e	s	h		R	a	n	i
--	---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---

%18s

R	A	j	u		R	a	j	E	s	h		R	a	n	i	
---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---	--

Maths Functions

# CPPM

C provides many common-used Mathematical functions in library <math.h>. The signatures of some of these functions are:

## Unit 4.

### 4.1. if statement

#### 4.1.1.Simple if statement

#### 4.1.2.if...else statement

#### 4.1.3.Nested if statement

### 4.2. while loop

### 4.3. do...while loop

### 4.4. for loop

### 4.5. break and continue statements

### 4.6. switch statement

C conditional statements allow you to make a decision, based upon the result of a condition. These statements are called as Decision Making Statements or Conditional

**sin(x), cos(x), tan(x), asin(x), acos(x), atan(x):**

Take argument-type and return-type of float, double, long double.

**atan2(y, x):**

Return arc-tan of y/x. Better than atan(x) for handling 90 degree.

**sinh(x), cosh(x), tanh(x):** hyper-trigonometric functions.

**pow(x, y), sqrt(x):**

power and square root.

**ceil(x), floor(x):** returns the ceiling and floor integer of

floating point number.

**fabs(x), fmod(x, y):**

floating-point absolute and modulus.

**exp(x), log(x), log10(x):** exponent  
and logarithm functions.

Statements.

# CPPM

C languages have such decision-making capabilities within its program by the use of following the decision making statements:

- If statement ○ [if statement](#)
  - [if-else statement](#) ○
  - [Nested if-else statement](#) ○
  - [else if-statement](#)
- [goto statement](#)
- [switch statement](#)
- Conditional Operator

**If Statement:** If a statement in C is used to control the program flow based on some condition, it's used to execute some statement code block if the expression is evaluated to true. Otherwise, it will get skipped. This is the simplest way to modify the control flow of the program. The if statement in C can be used in various forms depending on the situation and complexity.

- (1) Simple if Statement
- (2) if-else Statement
- (3) Nested if-else Statement
- (4) else-if Ladder

**(1) Simple if Statement:** if statement is used to check condition and make decision.

Simple if condition works for true condition only.

The basic format of Simple if statement is:

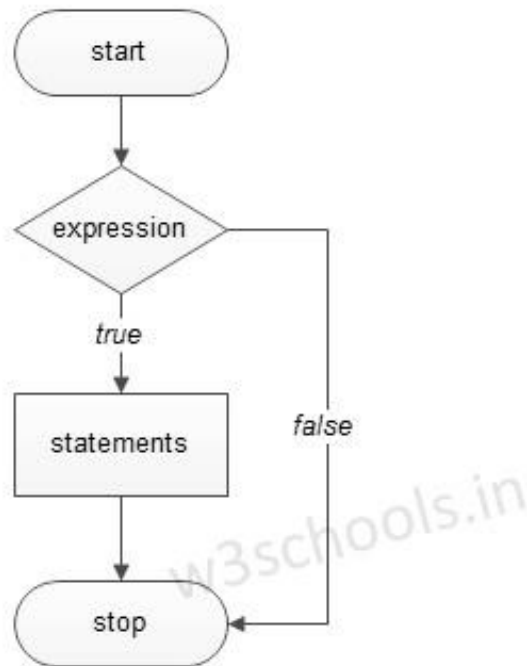
Syntax:

```
if(test_expression)
{
    statement 1;
statement 2;
    ... }
```

'Statement n' can be a statement or a set of statements, and if the test expression is evaluated to `true`, the statement block will get executed, or it will get skipped.

The flowchart of Decision-making technique in C can be expressed as:

# CPPM



```
#include<stdio.h>
main()
{
    int a = 15, b = 20;
    if (b>a)
    {
        printf("b is greater");
    }
}
```

**(2) if-else Statement:** If else statements in C is also used to control the program flow based on some condition, only the difference is: it's used to execute some statement code block if the expression is evaluated to true, otherwise executes else statement code block.

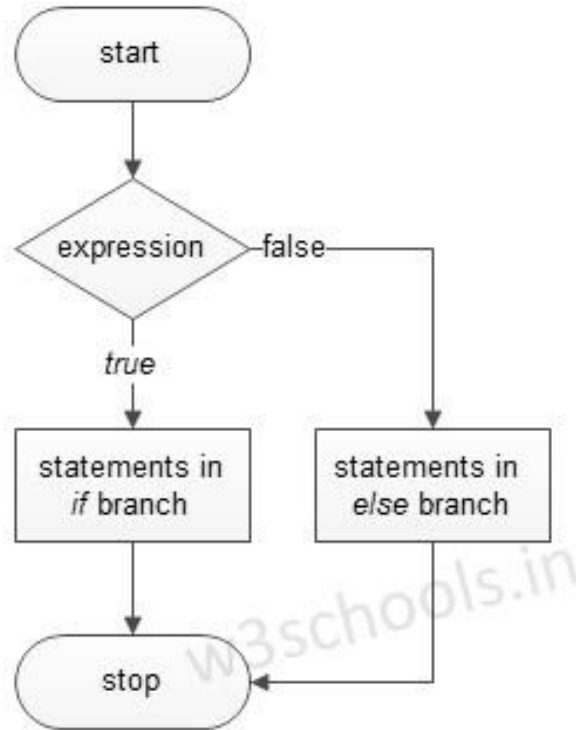
The basic format of if else statement is:

Syntax:

```
if(test_expression)
{
    //execute your code when condition is true
} else
{
    //execute your code when condition is false.
}
```

# CPPM

Figure - Flowchart of if-else Statement:



```
#include<stdio.h> main()
{
    int a = 15, b = 20;
    if (b>a)
    {
        printf("b is greater");
    }
    else
    {
        printf("a is greater");
    }
}
```

**(3) Nested if-else Statement:** Nested if allows to use conditional statements inside another conditional statement.

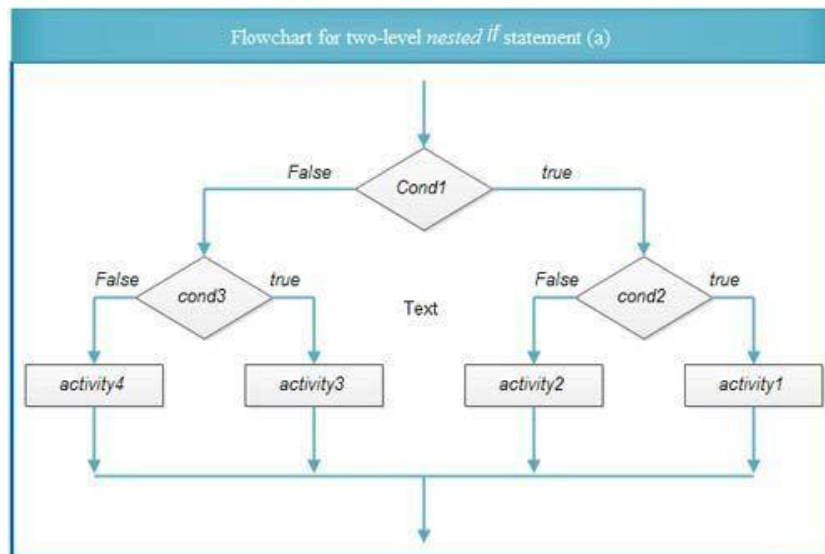
The basic format of Nested if else statement is:

Syntax:



# CPPM

```
if(test_expression one)
{
    if(test_expression two)
    {
        //Statement block Executes when the boolean test expression two
        is true.
    }
}
else
{
    //else statement block
}
```



## Example:

```
#include<stdio.h>
main()
{
    int x=20,y=30;

    if(x==20)
    {
        if(y==30)
        {
            printf("value of x is 20, and value of y is 30.");
        }
    }
}
```

# CPPM

```
}
```

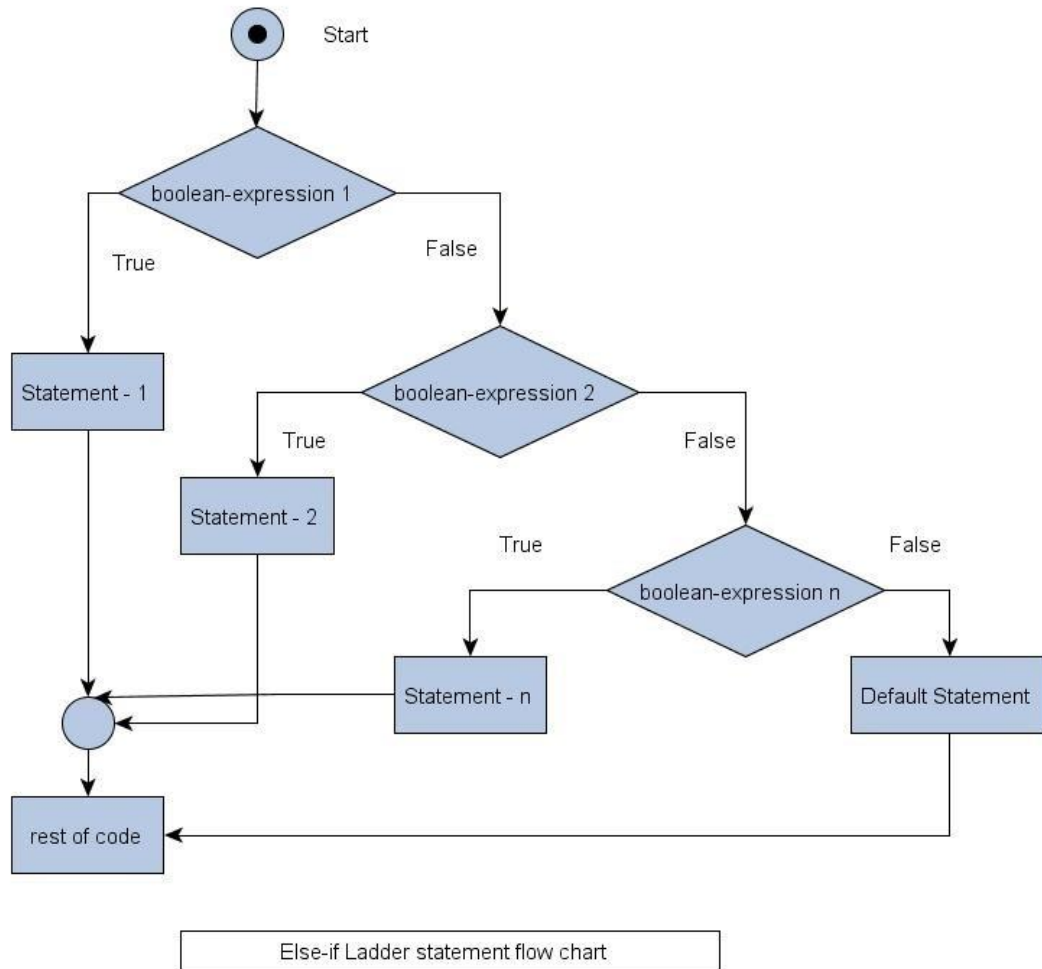
**(4) else-if Ladder:** else-if statements in C is like another if condition, it's used in a program when if statement having multiple decisions.

The basic format of else if statement is:

Syntax:

```
if(test_expression)
{
    //execute your code
}
else if(test_expression n)
{
    //execute your code
}
else
{
    //execute your code
}
```

# CPPM



Example:

# CPPM

```
#include<stdio.h> main()
{
    int a, b;

    printf("Please enter the value for a:");
    scanf("%d", &a);

    printf("\nPlease enter the value for b:");
    scanf("%d", &b);

    if (a > b)
    {
        printf("\n a is greater than b");
    }
    else if (b > a)
    {
        printf("\n b is greater than a");
    }

    else
    {
        printf("\n Both are equal");
    }
}
```

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

## Syntax

The syntax for a **switch** statement in C programming language is as follows –

```
switch(expression) {

    case constant-expression :
        statement(s);
        break; /* optional */

    case constant-expression  :
```

```
break; /* optional */

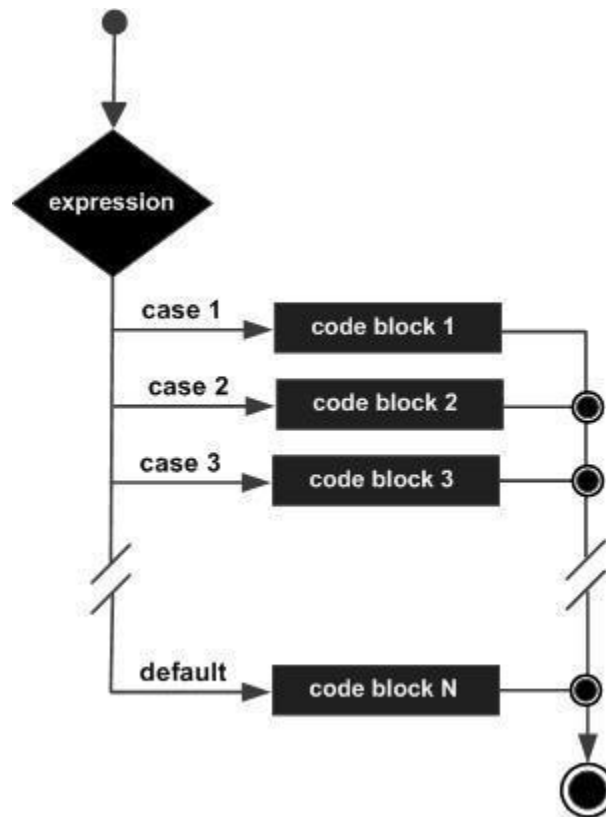
/* you can have any number of case statements */
default : /* Optional */ statement(s);
}
```

The following rules apply to a **switch** statement –

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

## Flow Diagram

# CPPM



**goto:** A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

**NOTE** – Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

## Syntax

The syntax for a **goto** statement in C is as follows –

- 1) Forward Jump

# CPPM

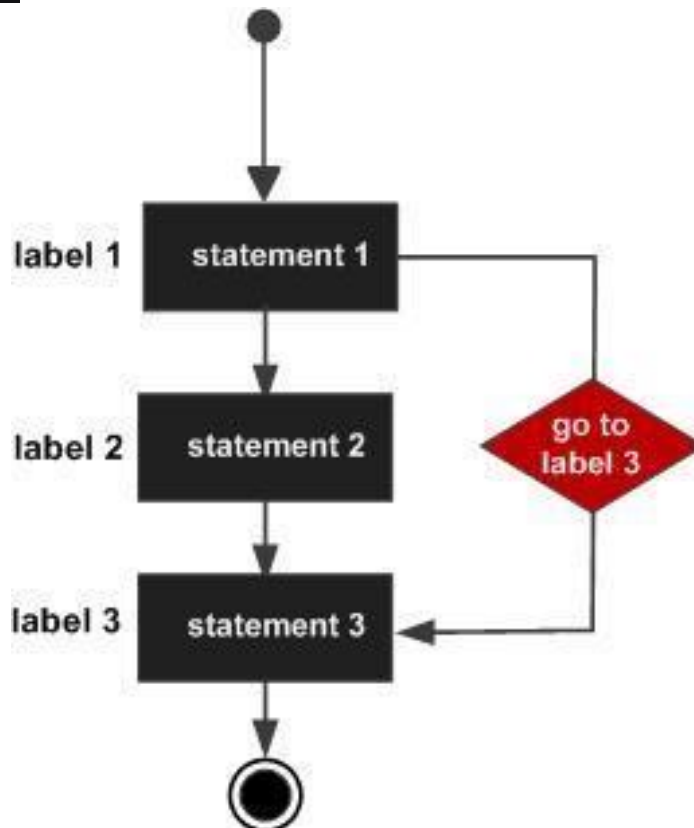
```
goto label;  
..  
.label:  
statement;
```

## 2) Backward Jump

```
Label:  
..  
.  
Goto Label1;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

## Flow Diagram



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

# CPPM

```
int main () {  
  
    /* local variable definition */  
    int a = 10;  
  
    /* do loop execution */  
    LOOP:do {  
  
        if( a == 15) {  
            /* skip the iteration */  
            goto LOOP;  
        }  
        a = a + 1;  
  
        printf("value of a: %d\n", a);  
        a++;  
    }while( a < 20 );  
  
    return 0;  
}
```

## **Loop**

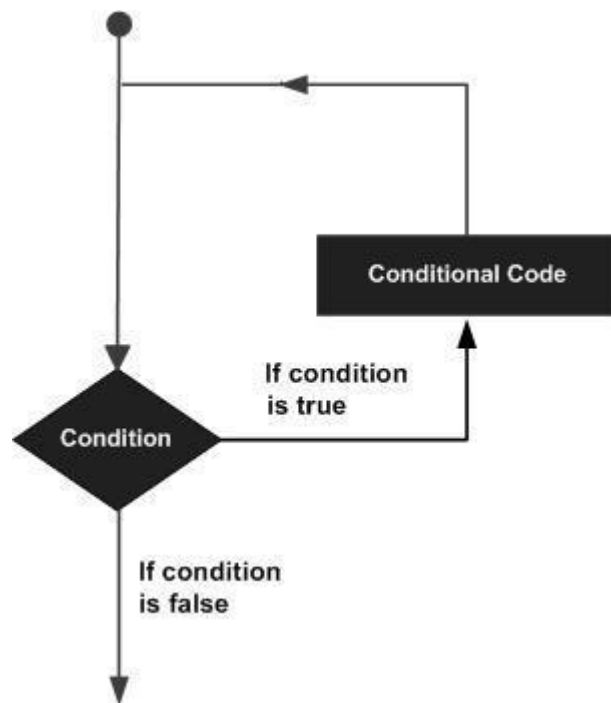
You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages –



# CPPM



C programming language provides the following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	<b><u>while loop</u></b> Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	<b><u>for loop</u></b> Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	<b><u>do...while loop</u></b> It is more like a while statement, except that it tests the condition at the end of the loop body.
4	<b><u>nested loops</u></b> You can use one or more loops inside any other while, for, or do..while loop.

# CPPM

## The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>

int main () {

    for( ; ; ) {
        printf("This loop will run forever.\n");
    }

    return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the `for(;;)` construct to signify an infinite loop.

- 1) **For loop:** A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. In this type of loop condition is checked first and then, code block is executed that is why this loop is called entry controlled loop.

### Syntax

The syntax of a **for** loop in C programming language is –

```
for ( init; condition; increment )
{
    statement(s); }
```

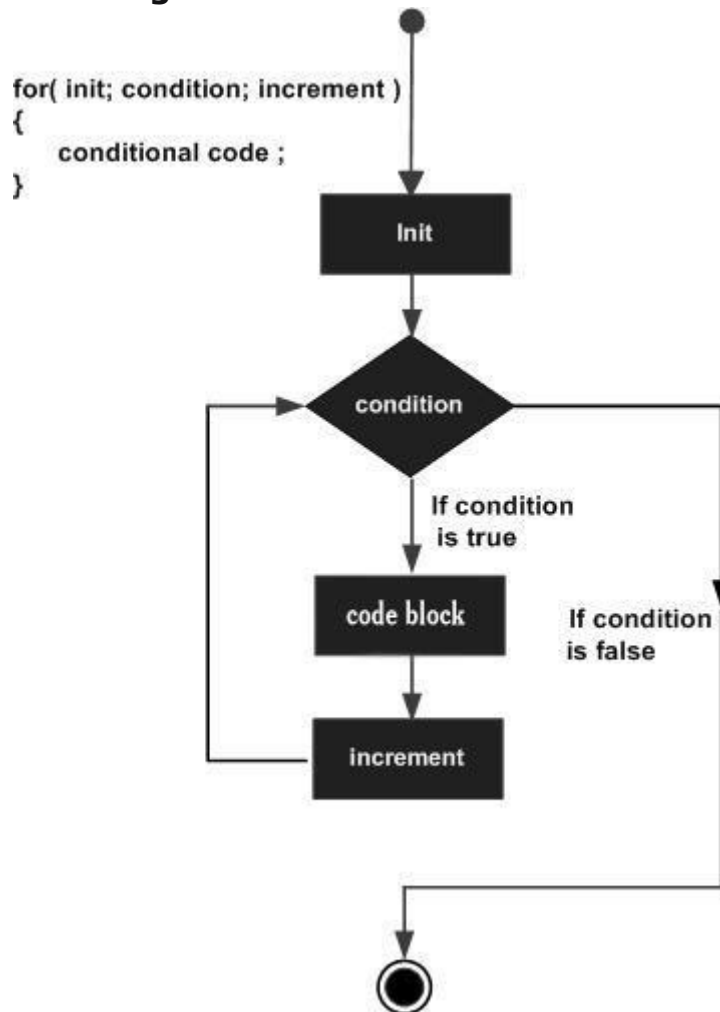
Here is the flow of control in a 'for' loop –

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.

# CPPM

- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

## Flow Diagram



# CPPM

```
#include <stdio.h>

int main () {
    int
a;

    /* for loop execution */    for(
a = 10; a < 20; a = a + 1 ){
printf("value of a: %d\n", a);
    }

    return 0;
}
```

2) **While Loop:** It is an entry controlled loop. In this type of loop condition is checked first and then, code block is executed.

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in C programming language is –

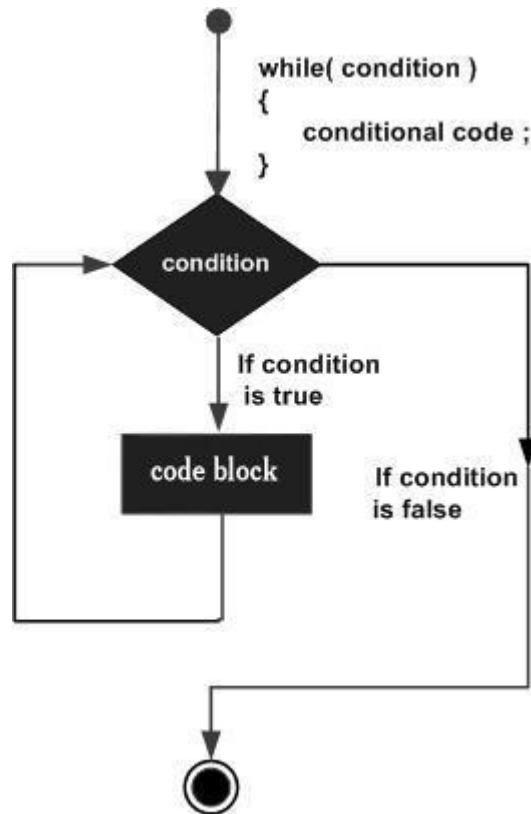
```
while(condition) {    statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

# CPPM

## Flow Diagram



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

```
#include <stdio.h>

int main () {

    /* local variable definition */    int a = 10;

    /* while loop execution */
    while( a < 20 ) {
        printf("value of a: %d\n", a);    a++;
    }

    return 0; }
```

- 3) **Do...while loop:** the **do...while** loop in C programming checks its condition at the bottom of the loop. It is exit controlled loop. A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

# CPPM

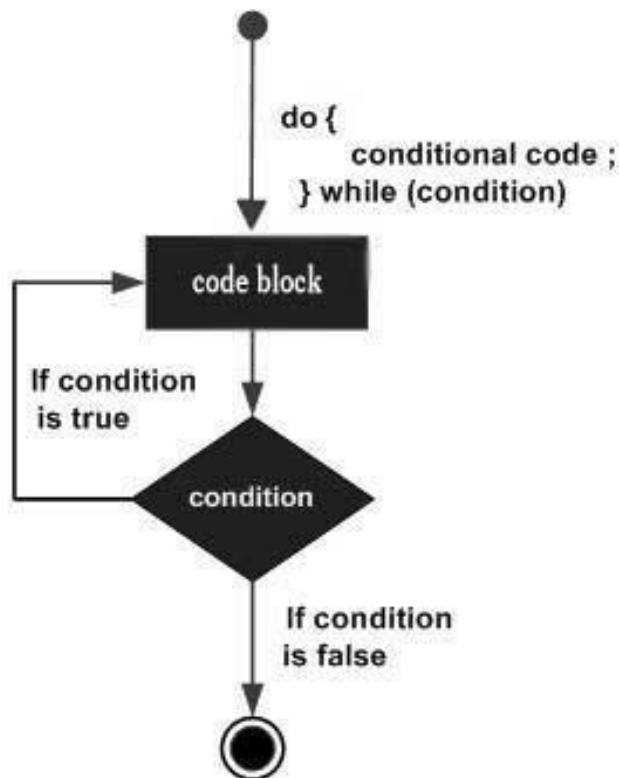
## □ Syntax

- The syntax of a **do...while** loop in C programming language is –

```
□ do {  
□     statement(s);  
□ } while( condition );
```

- Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.
- If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

## Flow Diagram



# CPPM

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do {
        printf("value of a: %d\n", a);
        a = a + 1;    }while( a < 20 );

    return 0;
}
```

**4) Nested Loop:** C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

## Syntax

The syntax for a **nested for loop** statement in C is as follows –

```
for ( init; condition; increment ) {

    for ( init; condition; increment ) {
        statement(s);
    }
    statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows –

```
while(condition) {

    while(condition) {        statement(s);
    }
    statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows –

# CPPM

```
do {  
    statement(s);  
    do  
    {  
        statement(s);  
    }while( condition );
```

```
}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

## Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 –

```
#include <stdio.h>  
  
int main () {  
    /* local variable definition */  
    int i, j;  
  
    for(i = 2; i<100; i++) {  
        for(j = 2; j <= (i/j); j++)  
        {  
            if(!(i%j)) break; // if factor found, not prime  
        }  
        if(j > (i/j)) printf("%d is prime\n", i);  
    }  
  
    return 0;  
}
```



## Unit 5. Arrays

### 5.1. One Dimensional Arrays

### 5.2. Sorting using One Dimensional Arrays

### 5.3. Concept of Two Dimensional Arrays

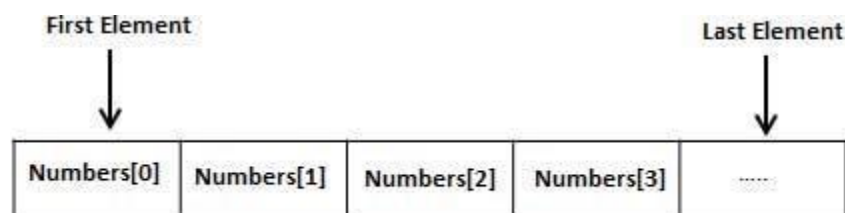
### 5.4. String- Array of characters

### 5.5. String Manipulation

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

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

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



Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and

# CPPM

located in separated memory location. Each elements of an array share a variable but each element having different index number known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

**ADVANTAGES:** array variable can store more than one value at a time where other variable can store one value at a time.

## Declaring Arrays

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

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

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

```
double balance[10];  
int arr[100]; int  
mark[100];  
int a[5]={10,20,30,100,5};
```

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

## Initializing Arrays

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

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

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

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

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

# CPPM

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

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

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

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

## Accessing Array Elements

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

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

The above statement will take the 10<sup>th</sup> element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

# CPPM

```
#include <stdio.h>

int main () {

    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;

    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */
    for (j = 0; j < 10; j++ ) {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

while initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializes.

For example:-       int  
marks[]={99,78,50,45,67,89};

# CPPM

If during the initialization of the number the initializes is less then size of array, then all the remaining elements of array are assigned value zero.

For example:-        `int`

```
marks[5]={99,78};
```

Here the size of the array is 5 while there are only two initializes so After this initialization, the value of the rest elements are automatically occupied by zeros such as

```
Marks[0]=99, Marks[1]=78, Marks[2]=0, Marks[3]=0,  
Marks[4]=0
```

Again if we initialize an array like `int array[100]={0};`

Then the all the element of the array will be initialized to zero. If the numbers of initializes are more than the size given in brackets then the compiler will show an error.

For example:-        `int`

```
arr[5]={1,2,3,4,5,6,7,8};//error
```

We cannot copy all the elements of an array to another array by simply assigning it to the other array like, by initializing or declaring as `int a[5] = {1,2,3,4,5}; int b[5]; b=a;` //not valid

**Two dimensional arrays:** Two dimensional array is known as matrix. The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are used.

Its syntax is:

```
Data_type array_name[row][column];
```

# CPPM

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row\*column**

Example:-

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

Total no of elements=row\*column is  $2*3 = 6$ . It means the matrix consist of 2 rows and 3 columns

For example:-

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
20	2	7	8	3	15
2000	2002	2004	2006	2008	2010

## Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example

```
int a[4][5];
```

## For reading value:-

```
for(i=0;i<4;i++)
{ for(j=0;j<5;j++)
    scanf("%d",&a[i][j]);
}
```

# CPPM

## For displaying value:-

```
for(i=0;i<4;i++)
{ for(j=0;j<5;j++)
    printf("%d",a[i][j]);
}
```

## Initialization of 2-d array:

2-D array can be initialized in a way similar to that of 1-D array.

For example:-

```
int mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};
```

These values are assigned to the elements row wise, so the values of elements after this initialization are

```
Mat[0][0]=11    Mat[1][0]=14    Mat[2][0]=17    Mat[3][0]=20
,
Mat[0][1]=12    Mat[1][1]=15,          Mat[3][1]=21
,          Mat[2][1]=18
Mat[0][2]=13
,
          Mat[1][2]=16    Mat[2][2]=19    Mat[3][2]=2
          ,          2
```

# CPPM

While initializing we can group the elements row wise using inner braces. for example:-

```
int mat[4][3]={ {11,12,13},{14,15,16},{17,18,19},{20,21,22}};
```

And while initializing, it is necessary to mention the 2nd dimension where 1st dimension is optional.

```
int mat[][3]; int  
mat[2][3];
```

```
int mat[][3];  
int  
mat[2][3];
```

 } invalid

If we **initialize an array** as

```
int mat[4][3]={ {11},{12,13},{14,15,16},{17}};
```

Then the compiler will assume its all rest value as 0, which are not defined.     Mat[0][0]=11,     Mat[1][0]=12,     Mat[2][0]=14,  
Mat[3][0]=17

Mat[0][1]=0,     Mat[1][1]=13,     Mat[2][1]=15     Mat[3][1]=0

Mat[0][2]=0,     Mat[1][2]=0,     Mat[2][2]=16, Mat[3][2]=0

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner.

We can also give the size of the 2-D array by using symbolic constant such as



# CPPM

```
#define ROW 2; #define  
COLUMN 3; int  
mat[ROW][COLUMN];
```

## String

Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character.

We can initialize the string as

```
char name[]={ 'j','o','h','n','\0' };
```

Here each character occupies 1 byte of memory and last character is always NULL character. Where '\0' and 0 (zero) are not same, where **ASCII** value of '\0' is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

J	o	H	N\0	
---	---	---	-----	--

The terminating NULL is important because it is only the way that the function that work with string can know, where string end.

String can also be **initialized** as; char  
name[]="John";

Here the NULL character is not necessary and the compiler will assume it automatically.

## String constant (string literal)

# CPPM

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character ('\0').

## Example – “m”

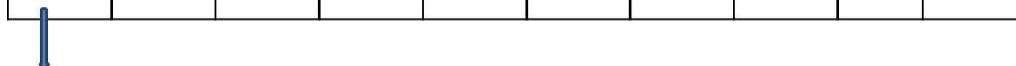
“Tajmahal”

“My age is %d and height is %f\n”

The string constant itself becomes a pointer to the first character in array.

Example-char crr[20]=“Taj mahal”;

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
T	A	J		M	A	H	a	l	\0



It is called base address.

**String library function:** There are several string library functions used to manipulate string and the prototypes for these functions are in header file “string.h”. Several string functions are **strlen()**

This function returns the length of the string. i.e. the number of characters in the string excluding the terminating NULL character.

It accepts a single argument which is pointer to the first character of the string. For example-

```
strlen(“suresh”);
```

It returns the value 6.

## In array version to calculate length:-

```
int str(char str[])
```

# CPPM

```
{ int i=0;
    while(str[i]!='\0')
    { i++;
    } return
    i;
}
```

## Example:-

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

void main()
{ char str[50]; print("Enter a
    string:"); gets(str);
    printf("Length of the string is %d\n",strlen(str));
}
```

## Output:

Enter a string: C in Depth

Length of the string is 8

**strcmp():** This function is used to compare two strings. If the two string match, strcmp() return a value 0 otherwise it return a non-zero value. It compare the strings character by character and the comparison stops when the end of the string is reached or the corresponding characters in the two string are not same.

strcmp(s1,s2) return  
a value:

<0 when s1<s2

=0 when s1=s2

>0 when s1>s2

The exact value returned in case of dissimilar strings is not defined. We only know that if s1<s2 then a negative value will be returned and if s1>s2 then a positive value will be returned.

## For example:

```
/*String comparison.....*/
```

# CPPM

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

void main()
{ char str1[10],str2[10];
  printf("Enter two strings:");

  gets(str1); gets(str2); if
  (strcmp(str1,str2)==0)
  { printf("String are same\n");
  } else
  { printf("String are not same\n"); }
}
```

**strcpy():** This function is used to copying one string to another string. The function strcpy(str1,str2) copies str2 to str1 including the NULL character. Here str2 is the source string and str1 is the destination string.

The old content of the destination string str1 are lost. The function returns a pointer to destination string str1.

Example:-

```
#include<stdio.h>
#include<string.h> void
main()
{ char str1[10],str2[10]; printf("Enter a string:");
  scanf("%s",str2); strcpy(str1,str2); printf("First
  string:%s\t\tSecond string:%s\n",str1,str2);
  strcpy(str,"Delhi"); strcpy(str2,"Bangalore");
  printf("First string :%s\t\tSecond string:%s",str1,str2);
}
```

## strcat()

This function is used to append a copy of a string at the end of the other string. If the first string is "Purva" and second string is "Belmont" then after using this function the string becomes "PurvaBelmont". The NULL character from str1 is moved and str2 is added at the end of str1. The 2nd string str2 remains unaffected. A pointer to the first string str1 is returned by the function.

Example:-

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

# CPPM

```
void main()
{
char str1[20],str[20]; printf("Enter two strings:"); gets(str1);
    gets(str2); strcat(str1,str2); printf("First string:%s\t
    second string:%s\n",str1,str2); strcat(str1,"-one");
    printf("Now first string is %s\n",str1); }
```

## Output

Enter two strings: database First string: database second string: database  
Now first string is: database-one

### Long Questions

1. Write a short note on one dimensional array.
2. Explain two dimensional array in detail.
3. Explain string array in detail.
4. Explain string functions.

### Short Questions

1. What is use of strcmp() function.
2. What is multi dimensional array?