A Complete study material for.....

'C' PROGRAMMING

SPIRIT AND SPARK GROUP

Solution toward your queries.....

Prepared By- Sanghdeep Sanghratne(Founder)

B.E Computer Engineering ,Mumbai University

Dedicated to my God.....

Prepared By- Sanghdeep Sanghratne(Founder)

'C' PROGRAMMING

INDEX

Sr. No.	Topic Name
1	Introduction to Software and Programming
2	Introduction to 'C' Programming
3	Preparing and Running 'C' Program
4	Standard Input and Output Functions
5	Operators in 'C'
6	Control Flow Structure
7	Arrays
8	Strings
9	Pointers
10	Functions
11	Storage Classes
12	Structure and Union
13	Dynamic Memory Allocations
14	Files

CONTENTS

1. **INTRODUCTION TO SOFTWARE AND PROGRAMMING**
    - What is Software?
    - Types of Software
    - Introduction to Programming
        - What is program
        - Steps in program Development

2. **INTRODUCTION TO 'C' PROGRAMMING**
    - Introduction to 'C' Language
    - History of 'C' Language
    - Application of 'C'
    - Why Should we learn 'C'
    - Characteristics/Features of 'C' Language
    - Flavors of 'C' Language
    - Character Set
    - 'C' Tokens
    - Data Types in 'C'
    - Types of Qualifier/Modifier
    - Variables and Constants

3. **PREPARING AND RUNNING ' C' PROGRAM**
    - Planning a 'C' program
    - Basic structure of 'C' program
    - Rules to write 'C' program
    - Compiling and executing the 'C' program
    - Detecting and correction of errors
    - Step to write and run 'C' program

4. STANDARD INPUT AND OUTPUT FUNCTIONS

- Library functions
- printf ( ) function
- scanf ( ) function
- getchar ( ) function
- putchar ( ) function
- gets ( ) and puts ( ) functions

5. OPERATORS IN 'C'

- Introduction
- What is Operator and Operands?
- Types of operators
  - Assignment operators
  - Arithmetic operators
  - Compound Assignment operators
  - Increment and Decrement operators
  - Comparison/Relational operators
  - Logical operators
  - Bitwise operators
  - Conditional operators
  - Special operators
  - Operator precedence and associativity

6. CONTROL FLOW STRUCTURES

- Introduction
- Types of Control Flow Structure
  - Sequential Control Structure
  - Selection/Decision Control Structure
    - ⇒ Simple if statement
    - ⇒ if….else statement
    - ⇒ Nested if….else statement

⇒ if….else ladder statement

⇒ switch statement

- Repetitive/Iterative/Loop Control Structure
  - ⇒ while Loop
  - ⇒ do….while Loop
  - ⇒ for Loop
- Jump Statement
  - ⇒ break
  - ⇒ continue
  - ⇒ goto
  - ⇒ return

7. ARRAYS

- Introduction
- What is an array?
  - Declaration of Array
  - Initialization of Array
- Types of array
  - Single/one dimensional array
  - Two dimensional array
  - Multidimensional array

8. STRINGS

- Introduction
- What is the string?
  - Declaration of strings
  - Initialization of string
- String related library functions

9. POINTERS

- Introduction
- What is pointer?
  - Declaration of pointer
  - Initialization of pointer
- Pointer arithmetic
- Pointers and arrays
- Pointers and strings
- Benefits of using pointers

10. FUNCTIONS

- Introduction
- Types of functions
  - Built in functions/Library functions
  - User defined functions
- Parts of functional program development
- Types of function call
  - Call by value
  - Call by reference(Pointers with function)
- Function classification
- Recursion

11. STORAGE_CLASSESS

- Introduction
- Storage classes in 'C'
  - auto
  - extern
  - register
  - static

12. STRUCTURE AND UNIONS

- Introduction
- Declaring a structure
- Initializing a structure member
- Rules for initializing a structure
- Array of structure
- Passing structure to function
- Pointers with structure
- Nested/Sub structure
- union
- Difference between structure and union
- typedef
- enumeration

13. DYNAMIC MEMORY ALLOCATIONS

- Introduction
- Definition
- DMA related predefined functions

14. FILES

- Introduction
- What is file?
- Types of file
- File handling
- File operation functions in 'C'

1. INTRODUCTION TO SOFTWARE AND PROGRAMMING

SOFTWARE

- **What is software?**

The computer system is made up of two major components known as ***Hardware*** and ***Software***. The ***Hardware is set of the physical components*** *such as, keyboard, mouse, monitor, CPU, hard disk etc.*

The ***Software refers to a program or set of instructions that instructs a computer to perform some task.*** Software helps users to view and run the programs. Computer system cannot be used without Software.

- **Types of Software**

Software can be categorized as follows:

- **System Software**
- **Application Software**
- **Embedded Software**
- **Web Applications**
- **Artificial Intelligence software**

- **System Software**

System software is a ***set of one or more programs***, designed to ***control the operation*** and ***extend the processing capability*** of a computer system.

It is the bridge between the hardware and users. Without system software user cannot use the hardware.

Examples: Operating system, Compilers, Interpreters, Editors, File manager etc.

- **Application Software**

Application software is developed to do the ***particular business activities***, which is designed as per ***users need.***

These software are used for business process or business management.

Examples: Point -of-sale transaction processing, real- time manufacturing process control etc.

- **Embedded Software**

Embedded software is computer software, written to ***control machines or devices*** that are not typically thought of as computers.

It is typically specialized for the particular hardware that it runs on and has time and memory constraints. Embedded software resides ***within a product or system.***

Examples: ATM Machine, Washing machine, automobile etc.

- **Web Application**

A Web application is an application program that is ***stored on a remote server and delivered over the Internet*** through a ***browser*** interface.

It is created in a browser-supported programming language (such as the combination of JavaScript, HTML and CSS) and relies on a web browser to render the application.

Examples: online bank or web-based email program like Gmail, Hotmail, or Yahoo Mail, online shopping application etc.

- **Artificial Intelligence software**

Artificial intelligence (AI) is an area of computer science that emphasizes the creation of ***intelligent machines*** that work and react like ***humans***.

Some of the activities computers with artificial intelligence are designed for include:

⇒ Speech recognition
⇒ Virtual reality and Image processing.
⇒ Nonlinear control and Robotics
⇒ Problem solving

PROGRAMMING

- **What is Program?**

A Program is simply a ***set of instructions*** that tells a computer how to perform a particular task. Programs are developed using ***programming languages.***

Programming languages is the art of developing computer programs. Programming is rather like a recipe; a ***set of instructions*** that tells a cook how to make a particular dish. It describes the ***ingredients (the data)*** and the ***sequence of steps (the process)*** on how to mix those ingredients.

- **Programmer**

A person who writes a program using a programming language is called a programmer. Programmer's job is to convert an algorithm into set of instructions which understood by a computer.

The programmer should also ***test*** the program to see whether it is ***working properly or not?***

- **Steps in a Program Development**

Developing a program involves ***a set of steps*** which are mentioned as follows:

1. ***Define the problem***
2. ***Outline the solution***
3. ***Develop an algorithm***
4. ***Test the algorithm for correctness***
5. ***Code the algorithm***
6. ***Compile the code/program***
7. ***Run the code/program***
8. ***Test, Document and maintain/save the program***

Most of these steps are common to any problem solving task. Program Development (software development) may take several hours, days, weeks, months or years.

After development, customers will make use of the system. While in use the system needs to be maintained. The maintenance phase will continue for several months, several years or even several decades.

Therefore ***software development*** is not a onetime task; it is ***lifecycle*** where some of above steps are reformed repeatedly.

Now we will study the above steps in details:

1. Define the problem

First of all the problem should be clearly defined. The problem can be divided into three components:

⇒ **Input:** What do you have?

⇒ **Output:** What do you want to have?

⇒ **Processing:** How do you go from inputs to outputs?

Programmer should clearly understand "*what are the inputs to the problem*". "*What is expected as output*", and "*how to process to generate required outputs".*

Consider a program to be written to calculate and display the sum of entered two numbers.

- **Input :** Two numbers(in variable such as 'a' and 'b')
- **Output :** Addition(result stored in variable 'c')
- **Processing :** `c = a + b`

2. **Outline the solution**

The programmer should define:

⇒ The major steps required to solve the problem

⇒ Any subtasks

⇒ The major variables and data structures

⇒ Major control structures(e.g. sequence, selection, repetition) in the algorithm

⇒ The underlined logic

Consider the above mentioned example, in order to calculate sum:

- **Variables :** Required variables are ***a*** **,** ***b*** and ***c***
- **Processing Logic :** `c = a + b`

3. **Develop an algorithm**

The next step is to develop an algorithm that will produce the desired result. An algorithm is a segment of precise steps that describes exactly the tasks to be performed; and order in which they are to be carried out to solve a problem. Pseudo code (a structured form of English language) can be used to express an algorithm.

A suitable algorithm for above mentioned example would be:

```
1. Start
2. Declare Variables a, b and c
3. Input first number in variable a
4. Input second number in variable b
5. Processing the logic c = a + b
6. Display result ,which stored in variable c
7. End
```

4. **Test the Algorithm for correctness**

The programmer must make sure that the algorithm is correct. The objective is to identify major logic errors early, so that they may be easily corrected. Test data should be applied to each step, to check whether the algorithm actually does what is supposed to.

The example mentioned above can be easily check by submitting some values for variable 'a' and 'b' walking through the algorithm to see whether the resulting output is correct for each input.

5. **Code the Algorithm**

After all design considerations have been met and when the algorithm is finalized, code it using a suitable programming language.

6. **Compile the code/program**

The next step is to compile the program. At this stage the compiler converts the *source program code to binary code/object code/machine code* (***Refer section Compiling Process***).

While compiling source program code syntax error may be occurs. When the written program doesn't recognized by the programming language, those are called ***syntax errors.***

These errors occur mostly due to miss typed characters, symbols, missing punctuations etc. The errors occurred at compile time is also called as ***compile time errors.***

If there are no syntax errors the program gets compiled and it produces an executable object code.

7. **Run the code/program**

After compiling the source code, the compiler generates the ***object code***. It is ***executable code*** which generates the output after it's execution.

While running the program the ***few errors*** may be occurs, those errors are called ***runtime error***. Mostly these errors may occur due to some illegal operation performed in the program.

8. **Test, Document and Maintain the program**

Test the running program using test data to make sure program is producing ***correct output.*** During these phase some errors can be found. To detect and remove those errors algorithm can be checked again.

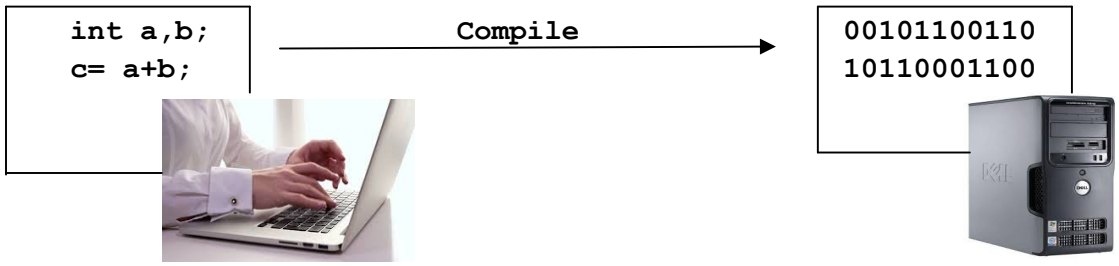
All the steps involved in developing the program algorithm and code should be ***documented for future reference***. Programmers should also maintain and update the program according to new or changing requirements.

- **Compiling Process**

After a program is developed using a programming language, it should be executed by the computer. Programmer writes program in human readable/understandable languages called high level languages.

However, the computer understands only binary ***1's and 0's***, which is referred as the ***machine language or machine code***. Therefore we need to convert the human readable programs into machine language before executing it. This conversion is achieved by special set of programs called ***compilers or interpreter***.

These programs convert ***the high-level language program***, into ***machine language programs.***



Conversion of the human readable code into machine code

Compiler translate a ***source program (human-readable)*** into an ***executable (machine-readable)*** program. Executable machine code produced by a compiler can be saved in a file (reffered as a executable file) and used whenever necessary.

Since the source program is already compiled, no compilation is needed again unless the source program is modified. To save the program on compilation time is an advantage that is gain from compiling and therefore these programs run much faster.

Programs written in programming languages such as FORTRAN, COBOL, C, C++ and PASCAL must be compiled before executing.

Each time the program runs, the interpreter converts high-level language instructions and input data to machine readable format and executes the program. *This process can be slower than the process which compile the source program before execution,* because of program needs ***conversion as well as execution at the same time.***

- **Programming Languages**

  Programming languages were invented to make programming easier. They became popular because they are much easier to handle than ***machine language.*** Programming languages are designed to be both ***high-level and general purpose.***

  A language is ***high-level*** if it is ***independent*** of the underlying hardware of a computer. It is general purpose if it can be applied to a wide range of situations.

There are ***more than two thousand*** programming languages and some of these languages are general purpose while others are suitable for specific applications.

Languages such as ***C, C++, JAVA, C# and Visual Basic*** can be used to develop variety of applications. On other hand ***FORTRAN*** was initially developed for numerical computing. ***Lisp*** for artificial intelligence, ***Simula*** for simulation and ***Prolog*** for natural language processing.

- **Generation of Programming Languages**

  There are following generation of programming languages:

  - **First Generation**
  - **Second Generation**
  - **Third Generation**
  - **Fourth Generation**

- **First Generation Programming Languages**

  In the early days, computers were programmed using machine language instructions that the ***hardware understood directly***. Program written using machine language belongs to first generation of programming languages.

  These programs were ***hardly human readable***, therefore understanding and modifying them was critical task.

- **Second Generation Programming Languages**

  Latter programs were written in a human readable version of machine code called ***Assembly language***. Each assembly language instruction directly maps into the machine language instruction (there is 1-to-1 mapping).

  Assembly language programs were ***automatically translated*** into machine language by a program called an ***assembler***. Writing and understanding Assembly language programs were easier than machine language programs.

  Programs developed in Assembly ***runs*** only on a ***specific type of computer.*** Further, programmers were required to have ***good sound knowledge*** about ***computer organization.***

- **Third Generation Programming Languages**

  With the introduction of third generation ***high-level*** languages were introduced. These languages allowed programmers to ***ignore the details of the hardware.***

The programs written using those languages were ***portable*** to more than one type of hardware. A ***compiler or an interpreter was used*** to translate the high-level-code to machine code.

Languages such as ***FORTRAN,COBOL, C*** and ***PASCAL*** belongs to the third generation.

- **Fourth Generation Programming Languages**

All the modern languages such as ***VISUAL BASIC, VB Script, JAVA, C#,*** and ***MATLAB*** belong to the ***fourth generation***. Programs written in these languages were ***more readable and understandable*** than the third generation languages.

They are much ***closer to natural languages***. ***Source cod***e of the programs written in these languages is much ***smaller*** than other generation of languages (i.e. a single high level language instruction maps into multiple machine language instructions).

However, programs developed in fourth generation languages generally ***do not utilize*** resources ***optimally***. They consume large amount of processing power and memory and they are ***generally slower than the programs developed using languages belonging to other generations.***

Most of these fourth generation programming languages support development of ***Graphical User Interfaces (GUIs)*** and responding to event such as movement of the mouse, clicking of mouse or pressing a key on the keyboard.

2. INTRODUCTION TO 'C' PROGRAMMING

- **Introduction to 'C' Language**

  'C' is general purpose structured programming language. There are mainly two types of languages, which are known as low level and high level languages. Low level languages interact with machine hardware. High-level languages are human understandable language like English.

  'C' allows to bring the gap between machine language and high language, This flexibility allows **'C'** to be used for system programming as well as for application programming. Therefore **'C'** is called a middle level language. Code written in 'C' language is very portable i.e. software written on one type of computer can be adapted to work on another type.

  'C' is very powerful programming language as most of the part of UNIX (multi-user operating system) is written in 'C'. Although 'C' has five basic built-in data type, it is not strongly typed language as compared to high level languages. 'C' permits almost all data types conversion. It also allows direct manipulation of bits, bytes, words and pointers.

  The learning of 'C' language will encourage you to develop your own logic and will prepare you for the advance programming tools. ***'C' Programming*** is the first step if you would like to be a good programmer and also ideal for system-level programming.

- **History of 'C' Language**

  'C' is a programming language developed at AT & T's Bell Laboratories in USA in 1972. It was written by Dennis Ritchie.

  'C' is very popular programming language due to it's simplicity, reliability and easy to use facility. The language like BASIC, COBOL, and FORTRAN had there own area of applications, where BASIC was treated as beginner's language COBOL was used for commercial purpose and FORTRAN was used for all engineering applications.

  Before **'C'** an International Committee tried to combine feature such as, simplicity, reliability and easy to use facility to form **ALGOL**. However this language could not gain popularity because it was too abstract and too general.

  To reduce the abstractness and generality, in **1966** a new language was developed called **BCPL** (Basic Combined Programming Language) whose simplification is **CPL** (Combined Programming Language). It was written by **Martin**

Ritche. BCPL is a simple, Procedural and not typed language. But it had limitation to use with limited Resources.

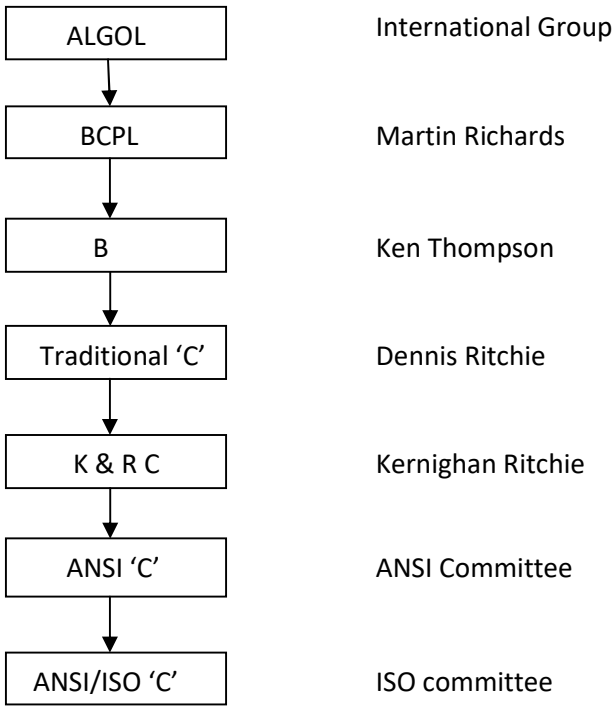
To overcome from these drawbacks **Ken Thompson** was evolved to design the **'B' language**, which he implemented the first Unix machines. However some limitations of previous languages were rewritten in this language.

From 1971, Dennis Ritchie did evolve to address the issues of 'B' language, which provides ability to the programmer to increment version of their programs. Dennis Ritchie 'Nudge' the letter 'B' to call the new language 'C'. This evolution is stabled to 1973, from which UNIX and UNIX SYSTEM utilities have been rewritten successfully in C.

Subsequently in 1978, Brian W. Kernighan documented very active language and finally published the book with Ritchie "The C Programming Language", Often called K&RC language as specified in the first edition of this book.

After this ANSI committee added many features to it, so it was ANSI C. At last ISO committee took it for improvements, so it was ISO C.

Following diagram shows the evolution of 'C' language.



History of 'C' Language

- **Application of 'C'**

C's ability to communicate directly with hardware makes it a powerful choice for system programmers. In fact, popular operating systems such as UNIX & LINUX are written entirely in 'C'.

Additionally, even compilers and interpreters for other languages such as FORTRAN, PASCAL and BASIC are written in 'C'.

However, C's scope is not just limited to developing system programs. It is also used to develop any kind of application, including complex business ones.

The following is a partial list of areas where C language is used:

- *Embedded Systems*
- *System Programming*
- *Artificial Intelligence*
- *Industrial Automation*
- *Computer Graphics*
- *Space Research*
- *Image Processing*
- *Game Programming*
- *To writes virus and antivirus*
- *Various electronics & communication products etc.*

- **Why Should we learn 'C'**
  - ***Nature of 'C'***

'C' is structured programming language , which means that it allows you to develop programs using well defined control structure(we will learn in chapter 6) and provides modularity (breaking the task into multiple sub task that are simple enough to understand and to reuse).

'C' is often called a middle-level language because it combines the elements of low-level or machine language with high-level languages.

There are many reason why one should select 'C' language before gaining to advance programming language, such as C++, C#, Java. They are as bellows:

⇒ 'C' is efficient, strong, simple and reliable. Also it is easy to learn.
⇒ 'C' programs run faster than most programs written in other language.
⇒ The basic concepts and programming skills of 'C' are used in C++, C#, JAVA. So without learning basics of 'C' it's very difficult to understand these advance object oriented programming languages.

⇒ For attaching new hardware to the computer system we require programs called as device drivers. These device drivers are generally 'C' programs.

⇒ Another major application of 'C' is 3D games where we need powerful graphical interface and fast speed.

- **Characteristics/Features of 'C' Language**

'C' has following characteristics which makes 'C' popular.

- ***General purpose Programming Language:***

It is general purpose programming language. It is usually called "System Programming Language" but equally suited to writing a variety of applications.

- ***Middle Level Language:***

As a middle level language it bridges elements of high level language with functionality of assembly language.

- ***Structured Programming:***

'C' is very much suited for structured programming. The programmers can easily divide a problem into a number of modules or functions.

- ***Simplicity:***

'C' is simple to use because of its structured approach .It has wide collection of inbuilt functions, keywords, operators and data types.

- ***Portability:***

This refers to the ability of a program to run in different environments. With the availability of compiler for almost all operating systems and hardware platforms, it is easy to write code on one system which can be easily ported to other system.

- ***Wide Acceptability:***

'C' is widely popular in the software industry. Its importance is not entirely derived from its use as primary development language but also because of its use as an interface language to some of the visual languages.

- ***Flexibility:***

'C' languages combine the convenience and portable nature of a high level language with the flexibility of low level language. It can be interfaced readily to other programming languages.

- ***Efficient Compilation and Execution:***

The process of compilation and execution of programs is quite fast in 'C' language as compared to other languages like BASIC (Beginner's All Purpose Symbolic Instruction Code) and FORTRAN (Formula translator).

- ***Modularity:***

'C' language programs can be divided into small modules with the help functions which are more helpful in understanding the programs.

- ***Clarity:***

The features like keywords, in-built functions and structures help to improve the clarity and hence understanding the program.

- ***High Availability:***

The software of 'C' language is readily available in market and can be easily installed on the computer.

- ***Easy Debugging:***

The syntax errors can be easily detected by the C compiler. The error is displayed with the line number of the code and the error message.

- ***Memory Management:***

Various memory management in-built functions are available in 'C' language, which helps to save memory and hence improve efficiency of the program. e.g. malloc(), alloc() and calloc().

- ***Recursion:***

Recursion is a technique in which the function calls itself again and again until condition is achieved.

- ***Rich set of Library Functions:***

'C' has a rich set of library functions. These are the function that provides readymade functionality to the users. It also supports graphics programming.

- **Flavors of 'C' Language**

'C' compiler can be available in different flavors since the portability was extended under various operating systems due to the significance of system level programming. Based on the operating system the 'C' compiler is energized much more stronger since all the system resources are under the control of operating system. If the parent operating system is stronger and massive, then the 'C' compiler also supports the massive task and activities

E.g.

1.Turbo	***C/C++***	***Dos***
2.ANSI	***C/C++***	***Unix***
3.Microsoft	***C/C++***	***Windows***
4.Borland	***C/C++***	***Winsows***
5.Berkley	***C/C++***	***Unix-BSD***
6.pro	***C/C++***	***Oracle-Unix/Windows***
7.Dynamic	***C/C++***	***Linux/Micro Linux, RTOS***
8.Embedded	***C/C++***	***Linux, RTOS***
9.Micro	***C/C++***	***Micro Linux, RTOS***

- **Character Set**

The Character set is the fundamental raw material of any language and they are used to represent information. Like natural languages, computer language will also have well defined character set, which is useful to build the programs.

'C' character set includes:

- ***Letters/Alphabets***
  1. Uppercase letters: A, B, C….,Z
  2. Lowercase letters: a, b, c…..,z
- ***Digits :****0, 1, 2,…,9*
- ***Special Characters:*** *, . : ; ! " ^ # % & * ( ) { } [ ] < > | \ / _ ~ etc.*

The special characters and its purpose to use in source code are listed as bellows

Character	Purpose /Use/Meaning	Character	Purpose /Use/meaning
~	Tilde	*	Asterisk
@	At symbol	'	Apostophe
_	Underscore	"	Quotation mark
^	Caret	!	Exclamation mark
&	Ampersand	?	Question mark
(	Left parenthesis	|	Vertical bar
)	Right parenthesis	<	Less than
[	Left square bracket	>	Greater than
]	Right square bracket	=	Equal to
{	Left flower /opening brace	/	Slash
}	Right flower /closing brace	\	Back slash
%	Percent sign	:	Colon
+	Plus sign	;	Semicolon
-	Minus sign	,	Comma
#	Number sign	.	Dot operator
$	Doller sign	----------	----------

- ***Whitespace characters:***

Character	Purpose /Use/Meaning	Character	Purpose /Use/Meaning
\b	Blank space	\t	Horizontal tab
\v	Vertical tab	\r	Carriage return
\f	Form feed	\n	New line
\\	Back slash	\'	Single quote
\"	Double quote	\?	Question mark
\0	Null	\a	Alarm (bell)

- **Ersio*n Characters(Format Specifiers):***

Conversion character or format specifiers are used to provide the format for the value to be print. It has the conversion character/prefix '%' and data specifier and an optional field with number (i.e. not compulsory).

We will learn more about ***Formatted Input*** and ***Formatted Output*** in ***chapter-4, i.e. Standard input and ouput functions.***

Format specifiers and its purpose are listed as follows:

Character	Purpose /Use/Meaning
%d	Prints integer value
%f	Prints float and double values
%c	Prints character
%s	Prints string
%o	Prints octal value
%u	Prints unsigned integer
%x	Prints hexa decimal value
%e	Prints scientific notation
%g	Prints scientific notation or floating points
%i	Prints decimal integer
%lu	Long unsigned integer
%ld	Long signed integer

- ***ANSI Escape Sequence:***

Certain non-printing characters as well as the double quote(" "), the apostrophe('), the question mark(?), and the backslash(\) can be expressed in terms of escape sequences. An escape sequence always begins with a backward slash and is followed by one or more special characters.

For example, a line feed (LF), which is referred to as new line in C, can be represented as '\n' such as escape sequences always represent single characters, even through they are written in terms of two or more characters.

The following escape sequences allow special characters to be put into the source code.

- ***Backslash character constants:***

Back slash characters are used in output function. These are used to format the output. The formatting output means the way of displaying the messages or results on the screen.

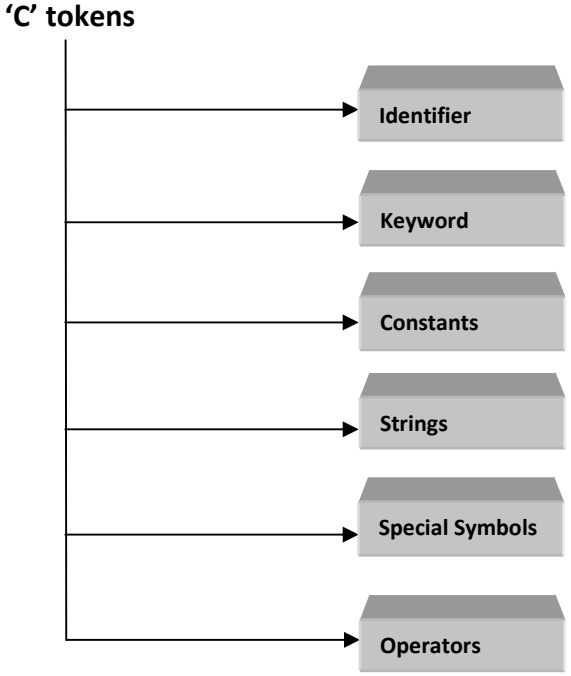
The following table displays the backslash characters:

Character	ASCII Value	Escape Sequence	Result
Null	000	\0	Null
Alarm(bell)	007	\a	Beep sound
Back space	008	\b	Moves previous position
Horizontal tab	009	\t	Moves next horizontal tab
New line	010	\n	Moves next line
Vertical tab	011	\v	Moves next vertical tab
Form feed	012	\f	Moves initial position of next page
Carriage return	013	\r	Moves beginning of the line
Double quote	034	\”	Present double quotes
Single quote	039	\’	Present Apostrophe
Question mark	063	\?	Present question mark
Back slash	092	\\	Present back slash
Octal number		\000	
Hexadecimal number		\x	

- **‘C’ Tokens**

The smallest individual unit identified by the compiler in a source program file is called as a ***‘token’***. It may be a single character, symbol, digit or sequence of characters to form a single item.

The tokens available in ‘C’ language are describe as bellows :



- **Identifiers :**

An identifier is a string of alphanumeric characters that begins with an alphabetic character or an underscore character which are used to represent various programming elements such as variables, functions, arrays, structures, unions, and so on.

Actually, an identifier is an user-defined word. There are total 63 characters to represents an identifier; which contains 52 alphabetic characters (i.e. both uppercase and lowercase alphabets), underscore character and 10 digits i.e. 0-9. The underscore character is considered as letter in identifiers. The underscore character is usually used in middle of an identifier.

⇒ **Rules for constructing identifier:**

1. The first character in an identifier must be an alphabet or an underscore and can followed only by any number alphabets or digits or underscore.

2. They must not begin with a digit.

3. Uppercase and Lowercase letters are distinct. that is, identifiers are case sensitive.

4. commas or blank spaces are not allowed within an identifier.

5. Keywords can not be used as an identifier.

6. Identifiers should not be of length more than 31 characters.

7. Identifiers must be meaningful, short, quick, easily typed and easily readable.

Following example shows valid and invalid identifier.

Valid Identifier: total, sum, average, _X, Y_ , num1, NUM_2

Invalid Identifier: 1X ----------> begains with a digit
char ----------> reserved word
x+y ----------> + is special character

There are two types of identifiers:

⇒ **External Identifier**

If the identifier is used in an external link process, then it is called as ***External Identifier***. These identifiers are also known as ***external names***; include function names and global variable names that are shared between source files. It has at least 63 significant characters.

⇒ **Internal Identifier**

If the identifier is not used in an external link process, then it is called as ***internal Identifier***. These identifiers are also known as ***internal names***; include the name of local variables. It has at least 31 significant characters.

- **Keywords**

'C' programs are constructed from a set of reserved words which provide control form libraries which perform special functions. The basic instructions are built up using a reserved set of words, such as ***main, for, if, while, default, extern, int, char*** etc. these reserved words are used only for giving commands or making statements. You cannot use default, for example, as the name of variable. If we try to do these, 'C' compiler will through the error.

Keywords have standard, predefined meaning in 'C'. These keywords can be used only for their intended purpose; They cannot be used as programmer-defined/ user-defined identifiers. Keywords are an essential part of a language definition. They implement specific features of the language.

A keyword is a sequence of characters that the 'C' compiler readily accepts and recognizes while being used in a program. Note that the keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier. The keywords are also called ***'Reserved Words'***.

⇒ Keywords are the word whose meaning has already been explained to the 'C' compiler and their meaning cannot be changed.
⇒ Keywords serve as basic building blocks for program statements.
⇒ Key word can be used only for their intended purpose.
⇒ Keyword cannot be used as user-defined variables.
⇒ All keyword must be written in lowercase.
⇒ There are total 32 keyword available in 'C'

The keywords with its categories are listed as bellows:

- **Data types**
  int, char, float, double
- **Decision making statement**
  if, else, switch, case, default
- **Derived classes**
  struct, union
- **Functions**
  void, return
- **Jump statement**
  goto, continue, break
- **Loop**
  for, while,do
- **Others**
  const, volatile, sizeof

- **Qualifiers**
  signed, unsigned, short, long
- **Storage classes**
  auto, extern, register, static
- **User-defined**
  typedef, enum

- **Constants**

  Constant in 'C' are fixed values which do not change during the execution of a program. Constants can be of any of the data types. 'C' supports several types of constants.

  We will discuss this term in details at the ending point ***(Variables and Constant)*** of these chapter.

- **Strings**

  String is collection/group of characters. When we write a word or sentence, it is treated as a string. E.g. sentence ***'SANGHDEEP'*** is called as string.

  We will discuss this term in details with declaration, initialization of string and all string related library function in ***chapter-8, i.e. Strings.***

- **Special symbols**

  We had already studied these term in details as ***'Special Character'*** in point ***'CHARACTER SET '***of these chapter.

- **Operators**

  An operator is a symbol or special character used in a program to manipulate data and variables as well as to perform certain mathematical or logical operations on data.

  We will discuss ***Operators and types of operators*** in details in ***chapter-5, i.e. Operators in 'C'.***

- **Data Types in 'C'**

  Data is differentiated into various types. A data type in a programming language is a set of data with values having predefined characteristics, such as integer, float and character. The language usually specifies the range of values for a given data type and how the values are processed by the computer and how they are stored. The storage size and the internal representation of these types are not specified in 'C' standard and may change from one compiler to another.

'C' supports a number of data types; if they are not enough programmer can also define their own data types.

The data types are categorized into following three major types:

- ***Primitive or Basic data types***
- ***User defined data type***
- ***Derived data type***

- **Primitive or Basic data types**

These are the fundamental data types supported by the 'C' language. These can be classified as:

1. *Integer type (**int**)*
2. *Floating point type(**float** and **double**) and*
3. *Character type(**char**)*

Each of these data types requires different storage capacities and has different range of values. Character (***char***) type is considered as an integer type and actual characters are represented based on their ASCII value.

Following table shows the memory size and range of values for primitive/basic data types.

Data Types	Memory size		Range of values
	Bit	***Byte***	
Char	8	1	-128 to +127
Int	16	2	-32768 to +32768
Float	32	4	-3.4e-38 to 3.4e+38 (accuracy up to 7 digits)
Double	64	8	1.7e-308 to 1.7e+308 (accuracy up to 15 digits)

- **User defined data type**

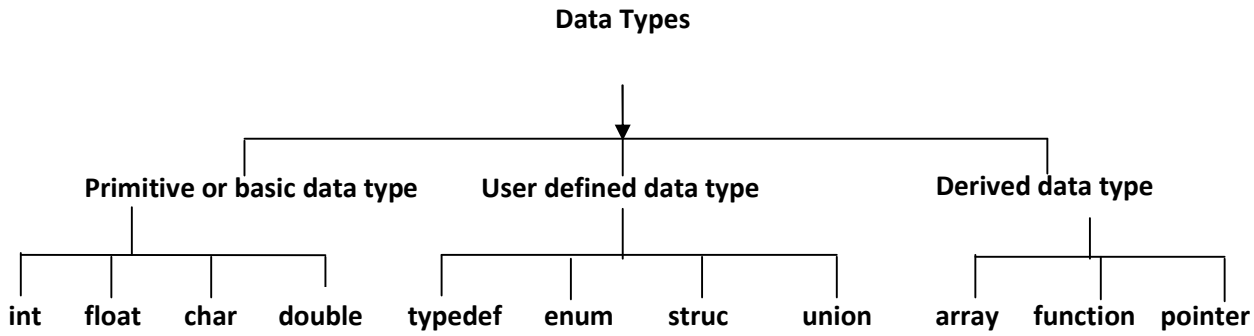
Based on the fundamental data types user can define their own data types. These includes type defined data type (using ***typedef*** keyword), enumerated types (using ***enum*** keyword), structures (using ***struct*** keyword) and unions (using ***union*** keyword).

All of these types can be created by combining several data types together.

- **Derived data type**

Programmer can derive data types such as ***arrays, functions*** and pointers.

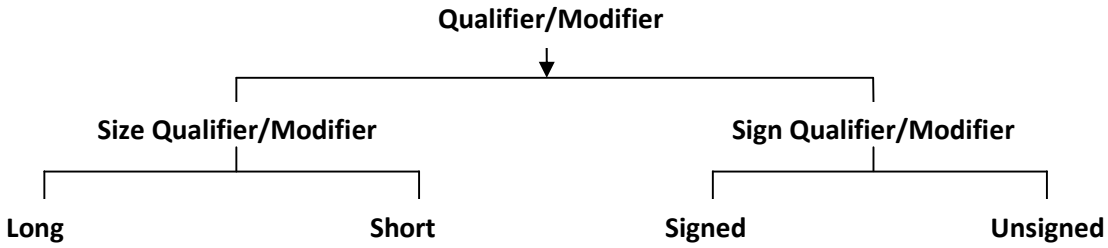
Following diagram shows the all data types.



- **Types of Qualifier/Modifier**

  The basic data types can be modified by adding special keywords called ***Qualifier/Modifier***,to produce new features or new types. A type modifier alters the meaning of base data type to yield a new data type.

  These modifiers can be classified into following categories:



- **Size Qualifier/Modifier:**

  These modifiers are used to modify the size of the basic data types. There are two size modifier i.e. ***Short*** and ***Long.***

- **Sign Qualifier/Modifier**:

  These modifiers are used to modify the sign of the basic data types. There are two size modifiers i.e. ***Signed*** and ***Unsigned.***

  Expect with ***void*** and ***float*** data type, the data type modifiers can be used with the basic data type's ***int, char, double.*** All of these ***modifiers*** can be applied to basic data type ***int***, ***signed*** and ***unsigned*** can be applied to basic data type ***char***, and only ***long*** can be applied to basic data type ***double.***

  - **Modification in 'int' data type:**

    All above modifiers can be used to modify basic data type int. When the basic data type is omitted in declaration then compiler assumes it 'int' by default.

See the following declaration which has same meaning.

```
long a;        /*base type omitted, assumed as int*/
long int a;    /*int already declared, does not need to assume*/
```

When an integer variable is declared it has short and signed as by default.

```
int a;              /* short & signed is default */
signed a;           /* int omitted, short is default */
signed int a;       /* short is default*/
short a;            /* int omitted, signed is default */
short int a;         /* signed is default */
short signed a;      /* int is default */
short signed int a; /* data types & modifier already declared */
```

Let's study what will happen when all these modifiers will be applied to basic data type 'int'. By default 'int a' variable is equal to 'short signed int a'.

Data Types	Memory size		Range of values
	Bit	***Byte***	
short signed int	16	2	-3278 to +32767
short unsigned int	16	2	0 to 65535
long signed int	32	4	-2,147,483,648 to +2,147,483,647
long unsigned int	32	4	0 to 4,294,967,295

- **Modification in 'char' data type:**

Char data type can be modified using two modifiers 'signed' and 'unsigned' by default a char variable is a 'signed char'.

Data Types	Memory size		Range of values
	Bit	***Byte***	
Char	8	1	-128 to +127
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255

- **Modification in 'double' data type:**

A double variable occupies 8 bytes by default. Only long modifier can be used to modify this base type .long modifier will increase size of this type by 2 bytes (16 bits).

Data Types	Memory size		Range of values
	Bit	*Byte*	
Double	64	8	1.7e-308 to 1.7e+308 (accuracy up to 15 digits)
Long double	80	10	3.4e-4932 to 1.1e+4932 (accuracy up to 15 digits)

- **Variables and Constants**
  - **Variables:**

Variables are the entities which can be changed at different times.

Programmer refers its name (identifier) so that it can be accessed during the execution of program. Programmer cannot use any of ***keywords*** as variable names. ***'A Variable name is an identifier or symbolic name assigned to the memory location where data can be stored and retrieved subsequently'.*** One variable can stores only one value at a time.

⇒ **Variables Declaration:**

In 'C' language, variable must be declared before it can be used. Variable can be declared before starting the function or within the function body. The variables are declared along with its data type, due to this compiler understands which types of data to be stored in it.

⇒ ***Rules for constructing Variables:***

1. It should begin with a letter or underscore and can contain only letters, underscore or digits. Special character or spaces are not allowed.
2. A variable name cannot be a keyword.
3. A variable name should not be of length more than 31 characters.

⇒ ***Syntax:***

data type space variable name;

According to variable declaration there are two types of variables.

1. Global Variable

2. Local Variable

1. Global variable:

If the variable declared before starting the function, then it is called ***global variable.*** It can be used in program anywhere, in any function.

2. Local variable:

It the variable declared within the function, then it is called ***local variable.*** It has limitation to use, we can use it only in that function where it has been declared.

⇒ ***Example:***

```
//TO ADD TWO NUMBERS
#include<stdio.h>
int a,b; //GLOBAL VARIABLE, BEFORE STARTING FUNCTION
main();
{
      clrscr();
      printf("Enter first number:\n");
      scanf("%d",&a);
      printf("Enter second number:\n");
      scanf("%d",&b);
      add();
      getch();
}
//FUNCTION USED FOR ADDITION
int add()
{
      int result; //LOCAL VARIABLE,DECLARED WITHIN FUNCTION
      result=a+b;
      printf("Addition is:\n%d",result);
      scanf("%d",&result);
      return 0;
}
```

In above program, variable **`int a,b`** are called global variable, because they are declared before function begins. Whereas variable **`int result`** is called local variable, because it is declared within the function **`int add()`**.

Multiple variables belonging to same data type can be declared as set of expressions by separating coma(,)sign.

e.g.

```
int a;
int b;
float c;
```

The above variables can also be declared as

```
int a,b; //VARIABLES BELONGING TO SAME DATA TYPE
float c;
```

⇒ **Variable Initialization :**

After Declaring a variable you have to initialized it. You can initialize a variable by giving it ***default value*** within the program or by taking value from user using **`scanf()`** ***function***.

An uninitialized variable can contain any garbage value therefore programmer must make sure all the variables are initialized before using tihem.

Example:

```
int a; //DECLARING VARIABLE a
a=5; //INITIALIZING VARIABLE a BY GIVING DEFAULT VALUE 5
```

You can initialize variable in same line with its declaration. In above example variable can be initialized as

```
int a=5; //DECLARING & INITIALIZING VARIABLE a BY GIVING DEFAULT VALUE 5
```

notice that the variable name appears to the left sign of the equal sign which is called the assignment operator. Only one variable name may appear.

Example:

```
c+D=a; //NOT ACCEPTABLE IN C LANGUAGE
a=c+D; //ACCEPTABLE IN C LANGUAGE
```

- **Constants:**

Constants in C are fixed values which do not change during the execution of program. Constants can be of any of the basic data types. C language supports several types of constants which are describes as follows.

- ***Numeric Constants***
  - *1. Integer Constants*
  - *2. Real or Floating-point Constants*
- ***Character Constants***
  - *1. Single Character Constants*
  - *2. String Character Constant*
- ***Declared Constants***
- ***Defined Constants***

- ***Numeric Constants:***

Numeric constant refers to the sequence of digits, it can be a whole number or fractional number and may have positive or negative sign.

There are two types of numeric constants:

1. ***Integer Constants:***

An Integer constants refers to a ***sequence of digits without a decimal point,*** it may have positive or negative sign.

Examples: 0
32767
-25

⇒ **Rules for constructing Integer Constant:**

1. An Integer Constant must have at least one digit.
2. It must not have decimal point.
3. It can either positive or negative.
4. If no sign precedence an Integer constant, it is assumed to be positive.
5. Commas or blank spaces are not allowed within an integer constant.
6. The range of Integer constant is from -32768 to +32767.

2. Real or Floating-Point Constants:

A real constant is a ***combination of a whole number followed by a decimal point and fractional part.***

The ***exponential representation*** is used when value is to small or to large. In this types of numbers representation there are two parts called ***mantissa*** and exponent. The part which comes ***before 'e'*** is known as ***mantissa*** and the part ***after 'e'*** is known as ***exponent.***

Example:

0.0
-0.7
+123.456
-3.2e-2
0.2e+7

In examples of ***-3.2e-2, -3.2*** is the ***mantissa*** and ***-2*** is the ***exponent.*** Similarly, In ***0.2e+7, 0.2*** is the ***mantissa*** and ***+7*** is the exponent.

⇒ ***Rules for constructing Real Constant:***

1. A real constant must have at least one digit.
2. It must have decimal point.
3. It can either positive or negative.
4. If no sign precedence an Real constant, it is assumed to be positive.
5. Commas or blank spaces are not allowed within a Real constant.
6. The range of Integer constant is from -3.4e38 to +3.4e38
7. The mantissa and exponent part should be separated by letter 'e'.

➢ ***Character Constants:***

A character constants can be ***single alphabet , sequence of alphabets or digit or a special symbol.*** The character constant is specified using left-pointed inverted commas (') or double quotes (" ").

There are two types of Character constants:

1. ***Single Character Constants:***

A single character constants is a ***single alphabet, a single digit or a single special symbol enclosed within single left –pointed inverted commas(') only.*** Notice that, double quotes (" ") is not valid to enclosed single character constant.

Examples: '5'
'S'
'*'

⇒ ***Rules for constructing Single Character Constant:***

1. The maximum length of a single character constant can be one character.

2. A single character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to left.

2. ***String Character Constants:***

A String character constants consists of a ***sequence of characters enclosed in*** double quotes (" "). A string character constant may contain combination of digits, alphabets, escape sequences and spaces.

Examples: "S"
'\o' ----------Represents null value
"SANGHDEEP"

In above example, String character constant "s" consists of character S and \o. However, a single character string constant does not have an equivalent integer value. It occupies two bytes, one for the ASCII code of S and another for NULL character with a value o, which is used to terminate all strings.

⇒ ***Rules for constructing String Character Constant:***

1. A string character constant consists of any combination of digits, alphabet, escaped sequences and spaces enclosed in double quotes (" ").

2. Every string character constant ends with a NULL character which is automatically assigned (before the closing double quotation mark) by compiler.

- ***Declared Constants:***

Declared constants are declared using the ***keyword const.*** With the const prefix the programmer can declare constants with a specific data type exactly as it is done with variables.

Example:

```
const float pi=3.14;
```

In above example we declared variable **pi** as **const** with data type float and initialized variable **pi** by assigning value **3.14** to it. During the execution of program the value of variable **pi** does not change because we declared it ass **const.*const*** allows us to create ***typed const.***

- ***Defined Constants:***

Programmer can define their own names for constants which are used quite often in a program. Without having to refer to a variable, such constant can be defined simply by using the ***#define pre-processor directive.*** This are called defined constants.

Example:

Following expression illustrate the use of **#define** ***pre-processor directive.***

```
#define pi=3.14;
```

If we try to assign any other value to **pi** compiler will generates the error. When we use **define** ***pre-processor directive*** to create constants then that does not required data type information.

3. PREPARING AND RUNNING 'C' PROGRAM

- **Planning a 'C' program**

It is necessary to do some planning before starting actual programming work. This planning will include the logic of the actual program. After logical approach of the program, the syntactic details of the language can be considered. This approach is often referred to as ***top-down programming.***

With the top-down approach we start with our top-level program, then divide and sub-divide it into different modules. This division process is known ***stepwise refinement.*** As we design each module, we will discover what kind of sub-modules we will need, then go on to main module and code it. After coding main module, we will need to codes smaller sub-modules, and groups them into a large module.

In the initial stage of program development program consists of only various elements that define major program components, such as function headings, function references etc. To describe the additional details/information about main components of program we use ***comments lines***. We will discuss more about comments lines in next section.

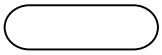
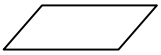
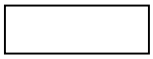
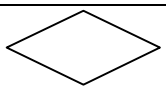
To develop program we use all program development steps which had been already we studied in ***chapter.1*** as ***Steps in program Development.*** But sometime those steps are not sufficient to develop large and complex program. In such situation we needs to develops ***flowchart*** for clear understanding of problems.

A Flowchart is a graphical representation of decisions and their results mapped out in individual shapes.

Flowchart first developed by ***Herman Goldstine*** and ***jon von Neumann*** in the 1940's.

Flowchart provides the step-by-step representation of algorithm for mapping out complex situations, such as programming code or troubleshooting problem.

Following table shows the different symbols, its names and functions which are used to draw a flowchart.

Symbol	Name	Function
	Start/End	An oval represents a start or end point
	Arrows	A line is a connecter that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

- **Basic structure of 'C' program**

Every programming language consists of certain major components, ***'C'*** language has the following major components:

- **Comment Lines:**

It is always good practice to comment your code whenever possible. Comments are useful for program maintenance.

Most of the programs needs to be modified after several months or years. In such cases programmer might not remember why he/she had written that program. In such cases comment lines makes programmers work easy in understanding the source code and the reason to writing them.

Compiler ignores all the comments and they do not have any effect on the executable code. There are two types of comment lines.

- ***Single Line Comments***

Single Line Comments starts with two slashes(//) and all the text until the line is considered as a comment

Single line comment can be applied only for single instruction of a program. Single line comment which applied for previous instruction does not working for next instruction. So that we use multiple/Block line comments for more than one instruction.

- ***Multiple/Block Line Comments***

Block line comments starts with characters (/*) and end with characters (*/). Any Text between those character is considered a block of comments.

These can be applied for single or multiple instructions in a program.

- **#include instruction:**

#include statement is used to include the ***standard library functions.*** These functions are written in the header files. To carry out the input and output operation we have need to include ***stdio.h file. stdio.h*** is ***standard input output header file*** which can be accessed by the program by including ***#include<stdio.h>*** instruction into the program.

To include ***mathematical readymade functions*** we have need to include ***math.h*** file. For console related functions ***conio.h*** is console input input output header file.

The functions are thus categorised into groups like math, console, stdio etc. Generally in most of ***'C'*** programs we include the statement #include<stdio.h>. Other header files are optional. They can be used as per users requirement. ***Pre-processor #include*** is always included at the beginning of the header file before starting the **`main()`** function.

- **Pre-Processor Directive:**

Pre-processor directive is ***#define,*** which we have already studied in ***chapter.2*** as ***defined Constant.*** It never terminates with semicolon(;). The ***# define*** is generally used to define ***constant values.*** These values are generally written in upper case letters, due to this it becomes easy to recognize the pre-processor directives.

Example: #define MAX 25.

- **Main Function:**

Every 'C' program must have **`main()`** function. Program would not run without **`main()`** function. The main function can be written in the following form:

```
1. Main()
2.void  main()
3.main(void)
4.void main(void)
5.int  (void)
6.int main()
```

The parenthesis '**()**' are used to indicate that main is a function and may pass the arguments. Arguments are the values in the form of variables which can be passed through the function. Keyword **void** indicates that no arguments passed through the function.

In above form of functions **Main()** indicates that argument may be passed through the **main()** function, and function may return the value. **void main()** indicates that **main()function** does not return any values. **main(void)** means no argument can be passed through the function. **void main(void)** indicates that value cannot be passed through the argument as well as function does not return any values.

int (void) means no argument can be passed through the **int** function. But **int main()** means function will return values. When **int** is specified with the **main()** function ,then we have need to write **return 0** as last statement of that function. The body of any function should be starts with '**{**' and ends with '**}**'.

Syntax of* `main()` *Function:

```
main()
{
     -----;
     Declarative or executable statements;

     ------;

 }
```

When we declare sub-function in a program, use the following syntax.

Syntax for* `Sub-`*Function:

```
Return_type space function name(argument list)
     {
          -----;
          Declarative or executable statements;

          ------;

          Return (expression);

          }
```

We will learn more about ***Function*** in details in ***chapter-10, i.e. Functions.***

Generally the structure of 'C' program is viewed as bellows:

```
Document Section
Link Section
Definition Section
Global Declaration Section
main() Function Section
        {
          Declarative part;
          executable part;
        }
Sub-Function Section
Function1
Function2
----------
----------
Function N
```

Structure of 'C' program

- Documentation section consists of the set of comment lines which may contain the name of program, author and other program related details.

- The next section is a link section which provides the instructions to the computer. Linker link functions from the programs to the library functions. Library is the readymade set of functions present in programming language. These functions are also called as readymade or library functions. So whatever data present in the program are cross checked with the library functions.

- Definition Section contains the Defined constants, so that the variable which defined as constant their value does not change during the execution of program.

- Global declaration section contains the global variables which can be shared by more than one function. It should be declared before **`main()`** function.

- Every program should contain **`main()`** function. The body of m function should be starts with ('{') and ends with('}'). The main function can be use with **`void`** or **`int`** according to its return type.

- Bellow **`main()`** function Sub-section part can be write.It is also called as ***User-Defined Function.*** These functions are generally written to perform specific task. The Sub-function section may contain at least one or more than one sub-function i.e. ***User-Defined Function.***

- **Rules to write 'C' program**

  - *Every program starts with* **`#include<stdio.h>`** *instruction* **.**
  - *Each and every program should contain* **`main()`** *function. Its body should start with* ***'{'****and ends with* ***'}'.***
  - *All instructions should be written in lower case letters, because* **'C'** *is case sensitive.*
  - *Before starting the main function Pre-processor directives should be declared.*
  - *Every* **'C'** *statement should be terminate with semicolon (;), Expect loops and function declaration.*
  - *All functions should be used with parenthesis i.e* **`()`** **.**
  - *The number of '{' should be equal to '}'.*
  - *Variable which are used in program should be declared in* **`main()`** *function or sub-function or before starting of the* **`main()`** *function.*
  - *Every* **'C'** *program should be saved with the extension* **'.c'.**

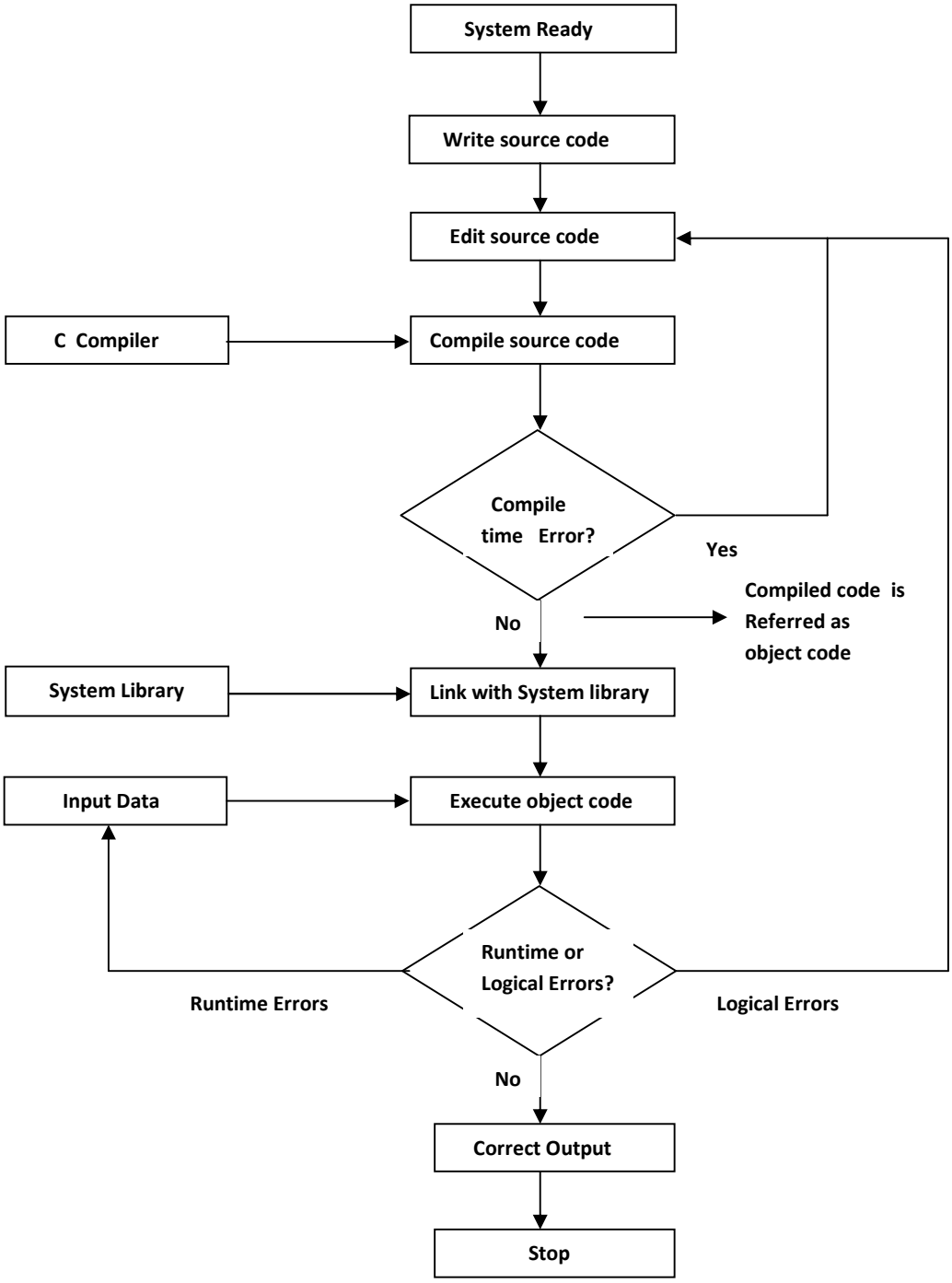
- **Compiling and executing the 'C' program**

  We have already studied ***Compiling Process*** in ***chapter.1***.

  All input and output functions are performed by an operating system. ***Operating system is interface between user and hardware.*** So the execution of program is done by operating system. In case of errors, the compiler shows the appropriate error messages and warning messages with indicating the lines which contains error in code.

  The program written in 'C' language is called source code, after compiling the source code it is called as object code. Once object code is generated by compiler, Linker Links the object code with the system library, it referred as executable code. When program is in executable state, this means it's ready to execute. ***It will show the correct output if there are no runtime errors, compile time errors or Logical errors***. We will study all these types of errors in details in next section i.e. **Detecting and correction of errors.**

Following flowchart shows the execution steps of 'C' program.



execution steps of 'C' program

- **Detecting and correction of errors**

To detect the errors in 'C' program it is necessary to compile the program which is written in editor window. To compile the program press **'Alt'** and **'F9'** key simultaneously or simply click on Compile option which appears on the top of editor window**.** Sometimes an error simply cannot be located. Some 'C' compilers include a debugger, which is a special program that facilitates the detection of errors in 'C' programs.

Errors are also known as bugs which may prevent the program to compile and run correctly as per expectation of programmer.

Basically there are three types of errors may occurred in 'C' program. Which are explained as bellow:

1. Runtime Errors
2. Compile time Errors
3. Logical Errors

1. Runtime Errors:

Runtime errors are those that occurs during the execution of program. It occurs due to some illegal operation performed in the program.

Examples:

- Dividing a number by zero.
- Trying to open a file which is not exists.
- Lack of free memory space.

It should be noted that occurrence of runtime errors may stops the execution of program thus to encounter these errors, a program should be written in such that it is able to handle such unexpected errors rather than terminating unexpectedly, it should be able to continue operating. ***This ability of program to handle unexpected errors is known as robustness.***

2. Compile time Errors:

Compile time errors are those errors that occurs at the time of compilation of the program.

Compile time errors are classified into two types as:

- ***Syntax Errors***
- ***Semantic Errors***

- ***Syntax Errors:***

When the rules of the 'C' programming language are not foolwed, the compiler will show syntax errors.

Examples:

Consider the statement,

```
int a,b :
```

The above statement will produce syntax error as the statement is terminated with '**:**' rather than '**;**'.

- ***Semantic Errors:***

Semantic errors are reported by compiler when the statements written in the program are not meaningful to the compiler.

Examples:

Consider the statement,

```
b+c=a;
```

In above statement we are trying to assign value of '**a**' in the value obtained by summation of '**b**' and '**c**' which has no meaning in **'C'** programming language.

The correct statement will be as,

```
a=b+c;
```

3. Logical Errors:

Logical errors are the errors in the output of the program. The presence of logical errors leads to undesired or incorrect output and are caused due to error in the logic applied in the program to produce the desired output.

Also logical errors could not be detected by the compiler and thus programmer has to check the entire code of program line by line.

Logical errors are often hard to find, so in order to find and correct errors of this type is known as logical debugging.

- **Step to write and run 'C' program**

1. *Run **tc.exe** or type **tc.exe** on command prompt or double click on the icon of if it's shortcut has created on desktop*
2. *After loading **tc.exe** an editor will get load. Click on file menu and select **'new option'.***
3. *Write/type the program in editor window.*

4. *Save the program by pressing **'F2'** key and specify the desired name to the program with **extension '.c'.***
5. *To compile the program press **'Alt'** and **'F9'** key simultaneously.*
6. *If compiler shows **warning:0 and Error:0** message window, it means **code is correct.** To escape this message window **press any key.***
7. *To run the program press **'Ctrl'** and **'F9'** key simultaneously.*
8. *If compiler **shows errors and warnings** in source code of program, it also shows the line number where error occurs. Go on to that line in editor window correct that code and **repeat this process from step-4 until warning and error becomes zero.***

4. STANDARD INPUT AND OUTPUT FUNCTIONS

- **Library functions**

'C' Language is accompanied by a number of standard library functions which carry out various useful tasks, such as input and output operations, string operations and all math operations are implemented by library functions. Library function is also known as built in function.

In order to use a library function, it is necessary to call the appropriate header file at the beginning of the program. At the time of compilation, header file inform the name, type and number and type of arguments of all of the functions which contained/included in the library.

A header file is called via the following pre-processor statement

```
#include<filename.h>
```

Where `filename` represent the name of the header file. Depending on program a programmer can include the following header file in the program.

- ***stdio.h***
- ***conio.h***
- ***string.h***
- ***stdlib.h***
- ***math.h***
- ***time.h***
- ***ctype.h***
- ***stdarg.h***
- ***signal.h***
- ***setjmp.h***
- ***locale.h***
- ***errno.h***
- ***assert.h***

The different library function along with its header file are listed as follows:

- ***stdio.h:***

This is standard input/output header file in which Input/Output functions are declared.

S.no	Function	Description
1	printf()	This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen
2	scanf()	This function is used to read a character, string, numeric data from keyboard.
3	getc()	It reads character from file
4	gets()	It reads line from keyboard
5	getchar()	It reads character from keyboard
6	puts()	It writes line to o/p screen
7	putchar()	It writes a character to screen
8	clearerr()	This function clears the error indicators
9	f open()	All file handling functions are defined in stdio.h header file
10	f close()	closes an opened file
11	getw()	reads an integer from file
12	putw()	writes an integer to file
13	f getc()	reads a character from file
14	putc()	writes a character to file
15	f putc()	writes a character to file
16	f gets()	reads string from a file, one line at a time
17	f puts()	writes string to a file
18	f eof()	finds end of file
19	f getchar	reads a character from keyboard
20	f getc()	reads a character from file

21	f printf()	writes formatted data to a file
22	f scanf()	reads formatted data from a file
23	f getchar	reads a character from keyboard
24	f putchar	writes a character from keyboard
25	f seek()	moves file pointer position to given location
26	SEEK_SET	moves file pointer position to the beginning of the file
27	SEEK_CUR	moves file pointer position to given location
28	SEEK_END	moves file pointer position to the end of file.
29	f tell()	gives current position of file pointer
30	rewind()	moves file pointer position to the beginning of the file
31	putc()	writes a character to file
32	sprint()	writes formatted output to string
33	sscanf()	Reads formatted input from a string
34	remove()	deletes a file
35	fflush()	flushes a file

- ***conio.h:***

  This is console input/output header file

S.no	Function	Description
1	clrscr()	This function is used to clear the output screen.
2	getch()	It reads character from keyboard
3	getche()	It reads character from keyboard and echoes to o/p screen
4	textcolor()	This function is used to change the text color

5	textbackground()	This function is used to change text background

- ***string.h:***

  All string related functions are defined in this header file

S.no	string functions	Description
1	**strcat(str1, str2)**	Concatenates str2 at the end of str1.
2	**strcpy(str1, str2)**	Copies str2 into str1
3	**strlen(strl)**	gives the length of str1.
4	**strcmp(str1, str2)**	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2.
5	**strchr(str1,char)**	Returns pointer to first occurrence of char in str1.
6	**strstr(str1, str2)**	Returns pointer to first occurrence of str2 in str1.
7	**strcmpi(str1,str2)**	Same as strcmp() function. But, this function negotiates case. “A” and “a” are treated as same.
8	**strdup()**	duplicates the string
9	**strlwr()**	converts string to lowercase
10	**strncat()**	appends a portion of string to another
11	**strncpy()**	copies given number of characters of one string to another
12	**strrchr()**	last occurrence of given character in a string is found
13	**strrev()**	reverses the given string
14	**strset()**	sets all character in a string to given character
15	**strupr()**	converts string to uppercase
16	**strtok()**	tokenizing given string using delimiter
17	**memset()**	It is used to initialize a specified number of bytes to null or any other value in the buffer
18	**memcpy()**	It is used to copy a specified number of bytes from one memory to another

19	**memmove()**	It is used to copy a specified number of bytes from one memory to another or to overlap on same memory.
20	**memcmp()**	It is used to compare specified number of characters from two buffers
21	**memicmp()**	It is used to compare specified number of characters from two buffers regardless of the case of the characters
22	**memchr()**	It is used to locate the first occurrence of the character in the specified string

- ***stdlib.h:***

  This header file contains general functions used in C programs

S.no	Function	Description
1	**malloc()**	This function is used to allocate space in memory during the execution of the program.
2	**calloc()**	This function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't
3	**realloc()**	This function modifies the allocated memory size by malloc () and calloc () functions to new size
4	**free()**	This function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.
5	**abs()**	This function returns the absolute value of an integer . The absolute value of a number is always positive. Only integer values are supported in C.
6	**div()**	This function performs division operation
7	**abort()**	It terminates the C program
8	**exit()**	This function terminates the program and does not return any value
9	**system()**	This function is used to execute commands outside the C program.
10	**atoi()**	Converts string to int
11	**atol()**	Converts string to long
12	**atof()**	Converts string to float

13	**strtod()**	Converts string to double
14	**strtol()**	Converts string to long
15	**getenv()**	This function gets the current value of the environment variable
16	**setenv()**	This function sets the value for environment variable
17	**putenv()**	This function modifies the value for environment variable
18	**perror()**	This function displays most recent error that happened during library function call.
19	**rand()**	This function returns the random integer numbers
20	**delay()**	This function Suspends the execution of the program for particular time

- ***math.h:***

  math.h header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.

S.no	Function	Description
1	floor ( )	This function returns the nearest integer which is less than or equal to the argument passed to this function.
2	round ( )	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer value greater than the argument.
3	ceil ( )	This function returns nearest integer value which is greater than or equal to the argument passed to this function.
4	sin ( )	This function is used to calculate sine value.
5	cos ( )	This function is used to calculate cosine.
6	cosh ( )	This function is used to calculate hyperbolic cosine.
7	exp ( )	This function is used to calculate the exponential "e" to the x^{th} power.
8	tan ( )	This function is used to calculate tangent.

9	tanh ( )	This function is used to calculate hyperbolic tangent.
10	sinh ( )	This function is used to calculate hyperbolic sine.
11	log ( )	This function is used to calculates natural logarithm.
12	log10 ( )	This function is used to calculates base 10 logarithm.
13	sqrt ( )	This function is used to find square root of the argument passed to this function.
14	pow ( )	This is used to find the power of the given number.
15	trunc.(.)	This function truncates the decimal value from floating point value and returns integer value.

- ***time.h:***

  This header file contains time and clock related functions

S.no	Functions	Description
1	setdate()	This function used to modify the system date
2	getdate()	This function is used to get the CPU time
3	clock()	This function is used to get current system time
4	time()	This function is used to get current system time as structure
5	difftime()	This function is used to get the difference between two given times
6	strftime()	This function is used to modify the actual time format
7	mktime()	This function interprets tm structure as calendar time
8	localtime()	This function shares the tm structure that contains date and time informations
9	gmtime()	This function shares the tm structure that contains date and time informations
10	ctime()	This function is used to return string that contains date and time informations
11	asctime()	Tm structure contents are interpreted by this function as calendar time. This time is converted into string.

- ***ctype.h:***

  All character handling functions are defined in this header file

S.no	Function	Description
1	**isalpha()**	checks whether character is alphabetic
2	isdigit()	checks whether character is digit
3	isalnum()	checks whether character is alphanumeric
4	isspace()	checks whether character is space
5	islower()	checks whether character is lower case
6	isupper()	checks whether character is upper case
7	isxdigit()	checks whether character is hexadecimal
8	iscntrl()	checks whether character is a control character
9	isprint()	checks whether character is a printable character
10	ispunct()	checks whether character is a punctuation
11	isgraph()	checks whether character is a graphical character
12	tolower()	checks whether character is alphabetic & converts to lower case
13	toupper()	checks whether character is alphabetic & converts to upper case

- ***stdarg.h:***

  Variable argument functions are declared in this header file

S.no	Header_file	Function	Description
1	stdarg.h	va_start()	This function indicates the start process of variable length argument list in a program
		va_arg()	This function is used to fetch the arguments from variable length argument list
		va_end()	This function indicates the end process of variable length argument list in a program

2	signal.h	signal()	It is used to install signal handler
		raise()	It is used to raise signal in a C program
3	setjmp.h	setjmp()	This function prepares to use longjmp() function
		longjmp()	It is used for non local jump
4	locale.h	setlocale()	It sets locale()
		localeconv()	It gets locale conventions
5	errno.h	errno()	This function sets errno value to 0 at the beginning of the program. This value is modified to other than 0 when an error occurs while any function call.
6	assert.h	assert()	This function gets an integer as paramenter. If this paramenter is 0, writes message to stderr. Then, terminates the program. If this paramenter is non 0, it does nothing.

- ***signal.h:***

  Signal handling functions are declared in this file

S.no	Header_file	Function	Description
1	stdarg.h	va_start()	This function indicates the start process of variable length argument list in a program
		va_arg()	This function is used to fetch the arguments from variable length argument list
		va_end()	This function indicates the end process of variable length argument list in a program
2	signal.h	signal()	It is used to install signal handler
		raise()	It is used to raise signal in a C program
3	setjmp.h	setjmp()	This function prepares to use longjmp() function
		longjmp()	It is used for non local jump
4	locale.h	setlocale()	It sets locale()
		localeconv()	It gets locale conventions

5	errno.h	errno()	This function sets errno value to 0 at the beginning of the program. This value is modified to other than 0 when an error occurs while any function call.
6	assert.h	assert()	This function gets an integer as paramenter. If this paramenter is 0, writes message to stderr. Then, terminates the program. If this paramenter is non 0, it does nothing.

- ***setjmp.h:***

  This file contains all jump functions

S.no	Header_file	Function	Description
1	stdarg.h	va_start()	This function indicates the start process of variable length argument list in a program
		va_arg()	This function is used to fetch the arguments from variable length argument list
		va_end()	This function indicates the end process of variable length argument list in a program
2	signal.h	signal()	It is used to install signal handler
		raise()	It is used to raise signal in a C program
3	setjmp.h	setjmp()	This function prepares to use longjmp() function
		longjmp()	It is used for non local jump
4	locale.h	setlocale()	It sets locale()
		localeconv()	It gets locale conventions
5	errno.h	errno()	This function sets errno value to 0 at the beginning of the program. This value is modified to other than 0 when an error occurs while any function call.
6	assert.h	assert()	This function gets an integer as paramenter. If this paramenter is 0, writes message to stderr. Then, terminates the program. If this paramenter is non 0, it does nothing.

- ***locale.h:***

  This file contains locale functions

S.no	Header_file	Function	Description
1	stdarg.h	va_start()	This function indicates the start process of variable length argument list in a program
		va_arg()	This function is used to fetch the arguments from variable length argument list
		va_end()	This function indicates the end process of variable length argument list in a program
2	signal.h	signal()	It is used to install signal handler
		raise()	It is used to raise signal in a C program
3	setjmp.h	setjmp()	This function prepares to use longjmp() function
		longjmp()	It is used for non local jump
4	locale.h	setlocale()	It sets locale()
		localeconv()	It gets locale conventions
5	errno.h	errno()	This function sets errno value to 0 at the beginning of the program. This value is modified to other than 0 when an error occurs while any function call.
6	assert.h	assert()	This function gets an integer as paramenter. If this paramenter is 0, writes message to stderr. Then, terminates the program. If this paramenter is non 0, it does nothing.

- ***errno.h:***

  Error handling functions are given in this file

S.no	Header_file	Function	Description
1	stdarg.h	va_start()	This function indicates the start process of variable length argument list in a program
		va_arg()	This function is used to fetch the arguments from variable length argument list
		va_end()	This function indicates the end process of variable length argument list

2	signal.h	signal()	It is used to install signal handler
		raise()	It is used to raise signal in a C program
3	setjmp.h	setjmp()	This function prepares to use longjmp() function
		longjmp()	It is used for non local jump
4	locale.h	setlocale()	It sets locale()
		localeconv()	It gets locale conventions
5	errno.h	errno()	This function sets errno value to 0 at the beginning of the program. This value is modified to other than 0 when an error occurs while any function call.
6	assert.h	assert()	This function gets an integer as paramenter. If this paramenter is 0, writes message to stderr. Then, terminates the program. If this paramenter is non 0, it does nothing.

- ***assert.h:***

  This contains diagnostics functions

S.no	Header_file	Function	Description
1	stdarg.h	va_start()	This function indicates the start process of variable length argument list in a program
		va_arg()	This function is used to fetch the arguments from variable length argument list
		va_end()	This function indicates the end process of variable length argument list in a program
2	signal.h	signal()	It is used to install signal handler
		raise()	It is used to raise signal in a C program
3	setjmp.h	setjmp()	This function prepares to use longjmp() function
		longjmp()	It is used for non local jump
4	locale.h	setlocale()	It sets locale()

		localeconv()	It gets locale conventions
5	errno.h	errno()	This function sets errno value to 0 at the beginning of the program. This value is modified to other than 0 when an error occurs while any function call.
6	assert.h	assert()	This function gets an integer as paramenter. If this paramenter is 0, writes message to stderr. Then, terminates the program. If this paramenter is non 0, it does nothing.

In this chapter we will studying the standard Input and Output functions such as **printf(),scanf(),getchar(),putchar(),gets()and puts()** functions. Because, these functions are mostly used in 'C' programs.

- **printf()function**

printf() function is used to print the ***character, string, float, integer, double, octal and hexadecimal values*** on to the output screen.

We use **printf()** function with ***'%d'*** format specifier to display the value of an integer variable.

Simillarly ***'%c'*** is used to display character, ***'%f'*** for float variable, ***'%s'*** for string variable, ***'%lf'*** for double and ***'%x'*** for hexadecimal variable.

To generate a newline, we use ***'\n'*** in **printf()** statement.

Note that, **C** language is case sensitive. So that **printf()** and **Printf()** are different from each others. It always be written in lower case.

⇒ **Program:**

```
//USE OF printf() STATEMENT
#include<stdio.h>
int main();
{
      Char ch= 'S';
      Char str[20]= "SPIRIT AND SPARK GROUP ";
      float flt= 10.715;
      int no= 786;
      double dbl=20.248637;
      clrscr();
      printf("Character is%c:\n",ch);
      printf("String is%s:\n",str);
      printf("Float value is%f:\n",flt);
      printf("Integer value is%d:\n",no);
```

```
        printf("Double value is%lf:\n",dbl);
        printf("Octal value is%o:\n",no);
        printf("Hexadecimal value is%x:\n",no);
        getch();
        return 0;
}
```

Output:

```
Character is: S
String is: SPIRIT AND SPARK GROUP
Float value is: 10.715
Integer value is: 786
Double value is: 20.248637
Octal value is: 1422
Hexadecimal value is: 312
```

You can see the output with the same data which are placed within the double quotes of **`printf()`** statement in the program except format specifiers, i.e.

'%d' got replaced by value of an integer variable **no**,
'%c' got replaced by value of character variable **ch ,**
'%f ' got replaced by value of float variable **flt ,**
'%lf ' got replaced by value of double variable **dbl,**
'%o' got replaced octal value which corresponding to integer variable **no ,**
'%x' got replaced by hexadecimal value which corresponding to integer variable **no.**

- **`scanf()` function**

`scanf()` function is used to read the ***character, string, numeric data*** from the keybord.

Consider the below program where user enters a character. This value is assigned to the variable **'ch'** and then displayed.

After this, user enters a string and this value is assigned to the variable '**str**' and then displayed.

⇒ **Program:**

```
//USE OF scanf() STATEMENT
#include<stdio.h>
int main();
{
        Char ch;
        Char str[100];
```

```
        Clrscr();
        printf("Enter any Character:\n");
        scanf("%c",&ch);
        printf("Entered Character is:%c\n",ch);
        printf("Enter any String:\n");
        scanf("%s",&str);
        printf("Entered String is:%s\n",str);
        return 0;
}
```

Output:

```
Enter any Character:
p
Entered Character is:p
Enter any String:
sanghdeep
Entered string is:sanghdeep
```

The format specifier ***'%d'*** is used in **`scanf()`** statement . So that, the value entered is received as an integer and ***'%s'*** is used for string.

Ampersand is used before variable name ***'ch'*** in **`scanf()`** statement as ***'&ch'*** .

It is just like in a pointer which is used to point to the variable.

- **`getchar()` function**

`getchar()` function is used to get or read the input ***(i.e. single unsigned character),*** at runtime.

If end-of-file or an error is encounterd **`getchar()`** return ***EOF***.

⇒ **Syntax:**

```
int getchar(void)
```

⇒ **Program:**

```
//USE OF getchar() FUNCTION
#include<stdio.h>
int main();
{
        Char ch;
        Clrscr();
        printf("Enter any Character:\n");
        ch=getchar();
        printf("Entered Character is:%c\n",ch);
```

```
        return 0;
}
```

Output:

```
Enter any Character:
p
Entered Character is:p
```

In above program we declare the variable **'ch'** as char data type, and then get the value through the **getchar()** library function and store it in the variable **'ch'.** And then print the value of variable **'ch'.**

During the program execution, a single character is get or read through the **getchar()Function.**

The given value is displayed on the screen and the compiler wait for another character to be typed. If you press the enter key, you will see the entered character is printed through the **printf()Function.**

Note that, if you enter string then **getchar()Function** only reads the initial character of that string.

- **putchar() function**

putchar()Function displays a single unsigned character on the screen. If end-of-file or an error is encounterd **getchar()** return ***EOF***.

⇒ **Syntax:**

```
puchar(variable_name)
```

⇒ **Program:**

```
//USE OF putchar() FUNCTION
#include<stdio.h>
void main();
{
      Char ch;
      Clrscr();
      printf("Enter any Character:\n");
      ch=getchar();
      printf("Entered Character is:\n");
      putchar(ch);
      getch();
}
```

Output:

```
Enter any Character:
p
Entered Character is:p
```

In above program we declare the variable **'ch'** as char data type, and then get the value through the **getchar()** library function and store it in the variable **'ch'.** And then print the value of variable **'ch'.**

During the program execution, **putchar() Function** accept a value of variable **'ch'** through the parameter and displayed it on the screen.

Note that, if you enter string then **putchar() Function** only reads the initial character of that string.

- **gets() and puts() functions**
  - **gets() functions :**

gets() functions reads characters from the standard input **i.e. stdin** and stores them as a **'C'** string into variable until a newline character or the end-of-file is reached.

⇒ **Syntax:**

```
gets(variable_name)
```

⇒ **Program:**

```
//USE OF gets() AND puts() FUNCTION
#include<stdio.h>
#include<conio.h>
int main();//MAIN FUNCTION
{
      Char str[100];
      Clrscr();
      printf("Enter a string:");
      //GET STRING INPUT USING gets() FUNCTION
      gets(str);
      printf("Entered string is:");
      //PRINT STRING  USING puts() FUNCTION
      puts(str);
      //WAIT FOR OUTPUT SCREEN
      getch();
      //MAIN FUNCTION RETURN STATEMENT
      return 0;
}
```

Output:

```
Enter a string: SPIRIT AND SPARK
Entered string is: SPIRIT AND SPARK
```

While running the above program, whenever **gets()** Statement encounters then interrupt will be wait for user to enter some text on the screen. When user starts typing the characters, it will be copied into the variable **'str'** and when user enters newline character then process of accepting string will be stopped.

A terminating ***NULL*** character is automatically appended after the characters copied to string variable **'str'.**

gets() uses stdin(standard input output) as source, but it does not include the ending newline character in the resulting string and does not allow to specify a maximum size for string variable **'str'** (which can lead to buffer overflow).

- **puts() functions :**

The **puts()** writes a string (and then a new-line characters) to the stdout stream.

⇒ **Syntax:**

```
puts(variable_name)
```

⇒ **Program:**

```
//USE OF gets() AND puts() FUNCTION
#include<stdio.h>
#include<conio.h>
int main();//MAIN FUNCTION
{
      Char str[100];
      Clrscr();
      printf("Enter a string:");
      //GET STRING INPUT USING gets() FUNCTION
      gets(str);
      printf("Entered string is:");
      //PRINT STRING  USING puts() FUNCTION
      puts(str);
      //WAIT FOR OUTPUT SCREEN
      getch();
      //MAIN FUNCTION RETURN STATEMENT
      return 0;
}
```

Output:

```
Enter a string: SPIRIT AND SPARK
Entered string is: SPIRIT AND SPARK
```

The **`gets()` functions** returns a unsigned character value if there is no error else it will return **EOF**.

5. OPERATORS IN 'C'

- **Introduction**

  In this chapter we will learn about operators such as, what is operator? , what is operand? and types of operators in ***'C'*** language.

- **What is Operator and Operands?**
  - **Operator :**

    An ***operator*** is a symbol that tells the compiler to perform certain mathematical or logical manipulations. ***Operators*** are used in program to manipulate data and variables.
  - **Operands :**

    An ***operands*** are data items that operators act upon to evaluate expressions. The operands can be integer quantities, floating point quantities or characters.

    Examples: consider the following expressions –

```
a= 2+4;
b=2.5*a;
```

    In above expressions + , = and * symbols are the operators and a, b are operand which has specific values. Also value/data item 2 ,4 and 2.5 are the operands.

- **Types of Operators**

  'C' language supports the following types of operators:
  - **Assignment Operators**
  - **Arithmetic Operators**
  - **Compound Assignment Operators**
  - **Increment and Decrement Operators**
  - **Comparison/Relational Operators**
  - **Logical Operators**
  - **Bitwise Operators**
  - **Conditional Operators/Ternary Operators**
  - **Special Operators**

- **Assignment Operators**

  The assignment operator is used to assign a value to a variable. The equal sign (=) represents the assignment operator.

  The format of assignment statement is;

  Variable_name = expression;

Examples:

```
a= 7;       // value of variable a becomes 5
a=7+3;      // value of variable a becomes 15
a=5*b;      // value of variable a becomes 5*value of variable b
a=b+c;      // value of variable a becomes sum of b+c
a=(x*x+y*y)/2;
```

- **Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations in the same way that are used in algebra.

There are five main arithmetic operator in **'C'**, they are as :

Operator	Meaning	Description
+	Addition	Add two numbers
-	Subtraction	Subtract second operands from list
*	Multiplication	Multiply two operands
/	Division	Divide numerator by denominator
%	Modulus or Modulo	Reminder of division

Example:

```
//USE OF arithmetic operators
      #include<stdio.h>
      void main();
      {
      int a, b, add, sub, mul, div;
      clrscr();
      printf("\n Enter a:");
      scanf("%d",&a);    //read value in a
      printf("\n Enter b:");
      scanf("%d",&b);    // read value in b
      add = a+b; // calculate sum of a & b
      sub = a-b; // calculate subtraction of b from a
      mul = a*b; //calculate multiplication of a & b
      div = a/b; // calculate division of a by be
      printf("\n Calculations are:");
      printf("\n a + b =%d", add); // print value of add
      printf("\n a – b = %d", sub); // print value of sub
      printf("\n a * b = %d", mul); // print value of mul
      printf("\n a/b = %d", div); // print value of div
      getch();
      }
```

When we executes above program with **'a'** as **27** and **'b'** as **5** it will show output as bellow.

Output:

```
Enter a: 27
Enter b: 5
Calculations are:
a + b = 32
a – b = 22
a * b =135
a/b = 5
```

Using arithmetic operators, we can perform following types of operations.

1) Integer Arithmetic Operation

2) Floating Point Arithmetic Operations

3) Mixed Mode Arithmetic Operations

1) Integer Arithmetic Operation

When an arithmetic operation is performed on two whole numbers or integer numbers then such an operation is called as integer arithmetic. It always gives an integer result.

Observe the following program.

```
//Integer arithmetic operation
      #include<stdio.h>
      void main();
      {
      int a=27, b=5, add, sub, mul, div, rem;
      clrscr();
      add = a+b; // calculate sum of a & b
      sub = a-b; // calculate subtraction of b from a
      mul = a*b; //calculate multiplication of a & b
      div = a/b; // calculate division of a by be
      rem = a%b; // calculate modulus of a divided by b
      printf("\n Calculations are:");
      printf("\n a + b =%d", add); // print value of add
      printf("\n a – b = %d", sub); // print value of sub
      printf("\n a * b = %d", mul); // print value of mul
      printf("\n a/b = %d", div); // print value of div
      printf("\n a % b = % d", rem); // print value of rem
      getch();
      }
```

Output:

```
Calculations are:
a + b = 32
a – b = 22
a * b =135
a/b = 5
a % b = 2
```

You might think that ***a/b*** in above program will be **5.4**,but in this case, the operands used in the expression are of integer type then this expression will return an integer result.

In integer division the fractional part is truncated.

2) Floating Point Arithmetic Operation

When an arithmetic operation is performed on two real numbers or fraction numbers then such an operation is called floating point arithmetic. The floating point result can be truncated according to the requirement. ***The modulo/modulus(%)*** operator is not applicable for floating point arithmetic operands.

Following program shows it.

```
//Floating Point arithmetic operation
      #include<stdio.h>
      void main();
      {
      float a=27.0, b=5.0, add, sub, mul, div, rem;
      clrscr();
      add = a+b; // calculate sum of a & b
      sub = a-b; // calculate subtraction of b from a
      mul = a*b; //calculate multiplication of a & b
      div = a/b; // calculate division of a by be
      rem = a%b; // calculate modulus of a divided by b
      printf("\n Calculations are:");
      printf("\n a + b =%f", add); // print value of add
      printf("\n a – b = %f", sub); // print value of sub
      printf("\n a * b = %", mul); // print value of mul
      printf("\n a/b = %f", div); // print value of div
      getch();
}
```

Output:

```
Calculations are:
a + b = 32.000000
a – b = 22.000000
```

```
a * b =135.000000
a/b = 5.400000
```

3) Mixed Mode Arithmetic Operations

When one of operand is real and other is an integer and if the arithmetic operation is carried out on these operands, then it is called as mixed mode arithmetic.

If any one of them operand is of real type (floating type) then the result will always be real. Following program shows it.

```
//Floating Point arithmetic operation
#include<stdio.h>
void main();
{
int a;
clrscr();
float b;result;
clrscr();
printf("\n Enter a:");
scanf("%d",&a);    //read value in a
printf("\n Enter b:");
scanf("%f",&b);    // read value in b
result=a/b;
printf("\n  Result is:%f", result); // print value of
div
getch();
}
```

Give the input a=15 and b=10.0, it will shows the output as,

Output:

```
a:15
b:10.5
result is:1.500000
```

- **Compound Assignment Operators**

  Compound assignment operators are used to write the statement in short format. Compound assignment operator is the combination of assignment operators with major arithmetic operators.

  For example, instead of statement ***a=a+b;*** programmer may use the shorthand format ***a+=b;***

  The list of compound assignment operators and its meaning are as follows:

Operator	*Meaning*	*Description*	*Example*
+=	Addition assignment operator	Adds right operand to the left operands and assign result to left operand	***a+=b*** is same as ***a=a+b***
-=	Subtraction assignment operator	Subtracts right operand from the left operands and assign result to left operand	***a-=b*** is same as ***a=a-b***
*=	Multiplication assignment operator	Multiply right operand with the left operands and assign result to left operand	***a*=b*** is same as ***a=a*b***
/=	Division assignment operator	Divides left operand by right operand and assign result to left operand	***a/=b*** is same as ***a=a/b***
%=	Modulus assignment operator	Calculate modulus using two operands and assign the result to left operand	***a%=b*** is same as ***a=a%b***

Refer following program which shows use of ***compound assignment operators***.

```
//USE compound assignment operators
      #include<stdio.h>
      void main();
      {
      int num,fact=1;
      clrscr();
      printf("\n Enter Number:");
      scanf("%d",&num);     //read value in num

            for (int i = 1; i <= num; i++)
                  {
```

```
                Fact*=i;
            }

        printf("\n Factorial of %d is :%d",num,fact);
        getch();
        }
```

Output:

```
Enter Number:5
Factorial of 5 is :120
```

We use ***for loop*** in above program, it is ***repetitive control structure*** .We will learn more about it in next chapter i.e. ***Control Flow Structure***.

- **Increment and Decrement Operators**

  Increment and Decrement operators are the examples of ***unary operators*** in ***'C'.*** In ***'C'*** programming, unary operators having higher priority than the other operators. Unary operators are executed before the execution of the other operators.

  Example :

```
a=-75;
b=-a;
```

Above statement assign the value ***-75*** to variable ***'a'*** and the value ***75*** i.e. ***(-(-75))*** to variable ***'b'***. The minus sign in this statement is called unary operator because it takes just one operand. There is no ***unary +*** in ***'C'*** programming.

- ***Increment Operator***

  Increment operator is used to increment the current value of variable by adding integer1. Increment operator can be applied to only variables.

  Increment operator is denoted by **++**.

Types of Increment Operators:

In ***'C'*** programming we have two types of increment operators:

1) **Pre-Increment Operator**
2) **Post-Increment Operator**

1) Pre-Increment Operator

Pre-increment operator is used to increment the value of variable before using in the expression. In Pre-increment, firstly value is incremented and then used inside the expression.

For example:

```
a=5;
b=++a;
```

In above expression, value of variable ***'a'*** is ***5***, then value of variable ***'b'*** will be *'6'* because the value of variable ***'a'*** gets incremented before using it in expression.

Observe the following program.

```
//USE of pre-increment operators
#include<stdio.h>
void main();
{
int a,b;
clrscr();
printf("\n Enter value for a:");
scanf("%d",&a);    //read value in a
b=++a;
printf("\n After pre-increment value of a=%d",b);
getch();
}
```

Output:

```
Enter value for a:5
After pre-increment value of a=6
```

2) Post-increment Operator

Post-increment operator is used to increment the value of variable after execution of the expression.

For example:

```
a=5;
b=a++;
```

In above expression, value of variable ***'a'*** is ***5***, then value of variable ***'b'*** will be *'5';* because it uses old value until execution of the expression. After its execution value of variable ***'b'*** will be incremented.

Observe the following program.

```
//USE of post-increment operators
#include<stdio.h>
void main();
{
int a,b;
clrscr();
printf("\n Enter value for a:");
```

```
scanf("%d",&a);    //read value in a
b=a--;
printf("\n After post-increment value of a=%d",b);
getch();
}
```

Output:

```
Enter value for a:5
After post-increment value of a=5
```

- ***Decrement Operator***

Decrement operator is used to decrement the current value of variable by subtracting integer 1 from it. Decrement operator can be applied to only variables also.

Decrement operator is denoted by **--**.

Types of Decrement Operators:

In ***'C'*** programming we have two types of decrement operators:

1) **Pre-decrement Operator**
2) **Post-decrement Operator**

1) Pre-decrement Operator

Pre-decrement operator is used to decrement the value of variable before using in the expression. In Pre-decrement, firstly value is decremented and then used inside the expression.

For example:

```
a=5;
b=--a;
```

In above expression, value of variable ***'a'*** is ***5***, then value of variable ***'b'*** will be *'4'* because the value of variable ***'a'*** gets decremented before using it in expression.

Observe the following program.

//USE of ***pre-decrement operators***

```
#include<stdio.h>
void main();
{
int a,b;
clrscr();
printf("\n Enter value for a:");
scanf("%d",&a);    //read value in a
b=--a;
printf("\n After pre-decrement value of a=%d",b);
```

```
getch();
}
```

Output:

```
Enter value for a:5
After pre-decrement value of a=4
```

2) Post- decrement Operator

Post-decrement operator is used to decrement the value of variable after execution of the expression.

For example:

```
a=5;
b=a--;
```

In above expression, value of variable ***'a'*** is ***5***, then value of variable ***'b'*** will be *'5';* because it uses old value until execution of the expression. After its execution value of variable ***'b'*** will be decremented.

Observe the following program.

```
//USE of post-decrement operators
#include<stdio.h>
void main();
{
int a,b;
clrscr();
printf("\n Enter value for a:");
scanf("%d",&a);    //read value in a
b=a--;
printf("\n After post-decrement value of a=%d",b);
getch();
}
```

Output:

```
Enter value for a:5
After post-decrement value of a=5
```

- **Comparison or Relational Operators**

Comparison or relational operators are used to test the relationship between two variables. It is mostly used in decision making statements to decide an action in program.

These operators have low precedence than unary and arithmetic operators.

The list of comparison/relational operators and its meaning are listed as follows.

Operator	Meaning	Description
>	greater than	Check if operand on the left side is greater than operand on the right side
<	Less than	Check if operand on the left side is less than operand on the right side
>=	Greater than or Equal to	Check if operand on the left side is greater or equal to operand on the right side
<=	Less than or Equal to	Check if operand on the left side is less than or equal to operand on the right side
==	Equal to	Check if two operands are equal
!=	Not Equal to	Check if two operands are not equal

These relational/comparison operators are used to form logical expressions representing condition that are either ***true*** or ***false***.

The resulting expression will be of type integer, so that true is represented by integer value ***'1'*** and false is represented by integer value ***'0'.***

Example :

```
 5<8   Evaluate TRUE (1)
(5+8)>=6   Evaluate TRUE (1)
(3+5)>(1+3)   Evaluate FALSE (0)
 5!=8  Evaluate TRUE (1)
 5==3  Evaluates  FALSE (0)
```

Following program shows the use of relational/comparison operators .

Program :

```
//USE of comparison operators
#include<stdio.h>
void main();
{
int num1,num2;
clrscr();
      printf("\n Enter First number:");
      scanf("%d",&num1);     //read value in num1
      printf("\n Enter Second number:");
      scanf("%d",&num2);     //read value in num2
```

```
            if(num1>num2)
            {
printf("\n First number is greater than Second Number );
            }
            else if(num1<num2)
            {
printf("\n Second number is greater than First Number );

            }
            else

      printf("\Both Numbers are equal );
      getch();
      }
```

Output:

```
Enter First Nummber:7
Enter Second Number:7
Both Numbers are equal
```

- **Logical Operators**

Many times programmer need to test more than one condition at a time and want to make a decision upon result of it. The logical operators are used to combine two or more expressions.

The entire expression is called as logical expression which evaluates to ***TRUE(1)*** or ***FALSE(0)***.

'C' has following logical operators.

Operator	***Meaning***
&&	Logical AND operator
||	Logical OR operator
!	Logical NOT operator

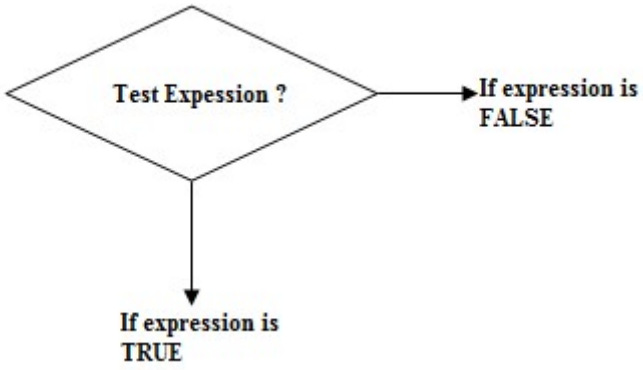
- ***Logical AND Operator***

Logical AND operator uses following decision table to make decision or to evaluate the expression.

Operand1	Operand2	Operand3
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

If all conditions are TRUE, then AND operator returns TRUE, else returns FALSE.

Following flowchart shows the working of logical AND operator.



- ***Logical OR Operator***

Logical OR operator uses the following decision table to evaluate the expression.

Operand1	Operand2	Operand3
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

If all conditions are FALSE, then Logical OR operator returns FALSE otherwise TRUE.

- ***Logical NOT Operator***

The logical NOT operator takes a single expression and reverse the value of expressions result. i.e. if the expression is true , the logical NOT(!) operator evaluates to FALSE and vice-versa.

Example :

!(7<11)

Evaluates to FALSE since 7 is less than 11

Following program shows the use of Logical Operators.

```
//USE of Logical operators
#include<stdio.h>
void main();
{
int num;
clrscr();
printf("\n Enter number:");
scanf("%d",&num);    //read value in a
      if(num>0 && num!=0 ||num!<0)
      {
printf("\n Number is positive  ");

      }
      else if(num<0 && num!=0 || num!>0  )
      {
printf("\n Number is negative ");

      }
      else if(num!>0 && num!<0 )
      {
printf("\n You enter number as zero ");
      }
getch();
}
```

Output:

```
Enter Number:0
You Enter number as zero
```

- **Bitwise Operators**

Bitwise operators are used to manipulate the data at bit level. The language in which user writes the program is high level language, but bitwise operator allows user to perform low level operations.

'C' supports following bitwise operators, their precedence is lower than arithmetic, relational and logical operators.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Right Shift
<<	Left Shift
~	One's Complement

Bitwise AND, OR and XOR operator perform logical AND, OR and XOR operations at bit level.

- **Bitwise AND Operation**

  Consider the following set of expressions;

```
int a,b,c;
a=12,b=8,c=a&b;
```

  the bitwise AND operation will be performed at bit level as follows:

```
 a=12-----> 00001100
*b=8 -----> 00001000
 -------------------
 C=a&b----> 00001000
```

Program :

```
// DEMO ON BITWISE AND operators
      #include<stdio.h>
      void main();
      {
      int a,b,c;
      a=12,b=8,c=a&b;
      clrscr();
      printf("\n Value  of a=%d and b=%d",a,b);
      printf("\n After performing Bitwise AND operation
      c=%d",c);
      getch();
      }
```

Output:

```
Value  of a=12 and b=8
After performing Bitwise AND operation c=8
```

In above example variable 'C' will hold the result of Bitwise AND operation between 'a' and 'b' which is `00001000`

- **Bitwise OR Operation**

Consider the following set of expressions;

```
int a,b,c;
a=12,b=8,c=a|b;
```

the bitwise OR operation will be performed at bit level as follows:

```
 a=12-----> 00001100
+b=8 -----> 00001000
 ------------------
 C=a|b----> 00001100
```

Program :

```
// DEMO ON BITWISE OR operators
      #include<stdio.h>
      void main();
      {
      int a,b,c;
      a=12,b=8,c=a|b;
      clrscr();
      printf("\n Value  of a=%d and b=%d",a,b);
      printf("\n After performing Bitwise OR operation
      c=%d",c);
      getch();
      }
```

Output:

```
Value  of a=12 and b=8
After performing Bitwise AND operation c=12
```

In above example variable 'C' will hold the result of Bitwise AND operation between 'a' and 'b' which is `00001100`

- **Bitwise XOR Operation**

Bitwise XOR operation can be used to invert the bits.For instance, write a program to convert a given character from **UPPERCASE** to **lowercase** and vice-versa.

ASCII(American Standard Code for Information Interchange) value of uppercase and lowercase character have difference of 32.

For example, In **ASCII** 'A' is represented by 65 where as 'a' is represented by 97. i.e. 97-65=32.

It can be represented in binary form as follows.

```
  65-----> 01000001
- 97-----> 01100001
  -------------------
  32-----> 00100000
```

Therefore any **ASCII** value X
OR with 32 will invert its case from upper to lower and lower to upper.

Program :

```
// DEMO ON XOR operators
      #include<stdio.h>
      void main();
      {
      Char var;
      clrscr();
      printf("\n Enter character to convert:");
      scanf("%d",var);// read charracter
      printf("\n Converted character is:%c",var^32);//invert
      entered character
      getch();
      }
```

Output:

```
      Enter character to convert:d
      Converted character is:D
```

- **Bitwise Shift Operation**

Shift operators allow shifting of bit either to left or right. We know that 8 is represented in binary as , `00001000` and when it is shifted by one bit to right we get `00000100` i.e. 4 observe the following operation.

```
8===> 00001000      after shifting 1-bit right
 ===> 00000100===>4
```

Similarly, when 8 is shifted by one bit to left, we get `00010000`, i.e. 16.

```
00001000<===8       after shifting 1-bit right
16<===00010000
```

The right shift operator(>>) shift an 8-bit number by one or more bits to right where as left shift operator(<<) shifts one or more bits to left.

Shifting binary number to left by one bit will multiply by 2 and shifting it to right by one bit will divide it by 2. After the shift operation final result should also be a 8-bit number, therefore any additional bits are removed and missing bits will be filled with zeros.

In general the right shift operator is used in the form,

Variable_name >> number of bits;

For example:

```
a>>2
```

In above example the value of variable 'a' is shift by 2 bits to right.

Left shift operator is used in the form,
Variable_name << number of bits;

For example:

```
a<<2
```

In above example the value of variable 'a' is shift by 2 bits to left.
Following program shows it.

Program :

```
// DEMO ON BITWISE SHIFT operators
      #include<stdio.h>
      void main();
      {
      int a,b,c;
      a=8;
      clrscr();
      printf("\n Value of a is %d",a);
      b=a>>1;//shifting 8 toward 1-bit right
      printf("\n After performing right shift operation
      a=%d",b);
      c=a<<1;//shifting 8 toward 1-bit left
      printf("\n After performing left shift operation
      a=%d",c);
      getch();
      }
```

Output:

```
Value of a is 8
After performing right shift operation a=4
After performing left shift operation a=16
```

- **One(1)'s Complement Operation**

One(1)'s Complement operator is also called as bitwise complement operator. It inverts the bits i.e. every 1 in binary representation of number will be inverted to 0 and every 0 is inverted to 1 to obtain the result.

For instance, 2 is represented in binary number as `00000010` when it inverted it will give result as `11111101`which is equal to -3.

3 has binary representation **00000011** .To obtain -3 we will take its 2's complement of 3 and add 1at LSB(Least significant Bit) to it to represent negative sign.

Following operation shows it

```
3===> 00000011   binary representation
 ===> 11111100   one's complement of 3
-------------------
      11111100
+             1  indicate negative sign
-------------------
      11111101 ===> 1's complement of 2
```

Program :

```
// DEMO ON Bitwise Complement operators
      #include<stdio.h>
      void main();
      {
      int a=2;
      clrscr();
      printf("\n Value of a is %",a);
      a=~a;//calculate 1'scomplement
      printf("\n After performing 1's complement operation
      a=%d ",a);
      getch();
      }
```

Output:

```
      Value of a is 2
      After performing 1's complement operation a=-3
```

- **Conditional or Ternary Operators**

Conditional operator is in combination of **'? :'.** This is called as ternary operator. It has three arguments, if the ***expression1*** is true; it returns ***expression2*** or if ***expression1*** is false, it will return ***expression3***.

Syntax:

Expression1? expression2: expression3;

e.g. `a=5<2?20:30;`

It will check condition 5<2 , the condition becomes false and so that it will return 30.

Program :

```
// DEMO ON conditional operator
#include<stdio.h>
void main();
{
int num1,num2,large;
clrscr();
printf("Enter first number:");
scanf("%d",&num1);
printf("Enter Second number:");
scanf("%d",&num2);
large=num1>num2?num1:num2;
printf("Large number is :%d",large);
getch();
}
```

Output:

```
Enter first number:5
Enter first number:7
Large number is :7
```

- **Special Operators**

There are certain operators which performs the special operations. These are really helpful. The special operators are ***comma, sizeof, pointer operator i.e. (& and *)*** and ***member selection operator such as ->***.

When comma is specified in the list, the list is evaluated from left to right. The comma operator has got lowest precedence as compare to remaining operators.

e.g. **var=(a=5,b=10,b-a)**

In above expression firstly 5 will be assigned to the variable 'a'. 10 will be assigned to the variable 'b'. And finally 'b-a' will be '10-5' i.e. 5.

So that final value of variable 'var' will be 5.

The size of operator is unary compile-time operator. It returns the length in the bytes of the variable.

Refer the following program.

Program :

```
// DEMO ON sizeof operator
#include<stdio.h>
void main();
{
float f;
```

```
clrscr();
printf("size of float data type is:%d byte\n",sizeof
f);
printf("size of double data type is:%d byte
\n",sizeof(double));
getch();
}
```

Output:

```
size of float data type is:4 byte
size of double data type is:8 byte
```

We will learn more about ***pointer operator*** and ***member selection operator*** in ***chapter-9, i.e. Pointers*** and ***chapter-12,i.e. Stucture and union.***

- **Operator precedence and associativity**

Precedence stands for priority. One expression may have multiple operators for the purpose of calculation. In this situation to evaluate the expression, compiler follows certain rules. This rule specify how to solve the expression and which operator will get priority in the group of operators.

According to priority set, the expression get evaluate or execute. Each operator is associated with its precedence.

The associativity is the feature which shows that expression evaluation of operators will take place from either left to right or right to left. This indicates the direction in which evaluation take place.

Following table shows the list of operator precedence with their associativity.

Operator	Description	Associativity	Rank
( )	Function call	Left to right	1
[ ]	Array element reference	Right to left	2
+	Unary plus	Right to left	2
-	Unary minus	Right to left	2
++	Increment	Right to left	2
--	Decrement	Right to left	2
!	Logical negation	Right to left	2
~	Ones complement	Right to left	2
*	Pointer reference	Right to left	2

Operator	Description	Associativity	Rank
&	Address	Right to left	2
Sizeof	Size of an object	Right to left	2
(type)	Type cast (conversion)	Right to left	2
*	Multiplication	Left to right	3
/	Division	Left to right	3
%	Modulus	Left to right	3
+	Addition	Left to right	4
-	Subtraction	Left to right	4
<<	Left shift	Left to right	5
>>	Right shift	Left to right	5
<	less than	Left to right	6
<=	less than or equal to	Left to right	6
>	Greater than	Left to right	6
>=	Greater than or equal to	Left to right	6
==	Equality	Left to right	7
!=	Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Right to left	10
&&	Logical AND	Right to left	11
||	Logical OR	Right to left	12
?:	Conditional expression	Right to left	13
=	Assignment operator	Right to left	14
*=/=%=	Assignment operator	Right to left	14
+=-=&=	Assignment operator	Right to left	14
^=|=	Assignment operator	Right to left	14
<<=>>=	Assignment operator	Right to left	14
,	Comma operator	Right to left	15

6. CONTROL FLOW STRUCTURE

- **Introduction**

  A program consists of statements or instructions, which are usually executed in sequence. Program can be much more powerful if we can control the order of execution of the program statements.

  The term ***flow of control*** refers to the order in which programs statements are executed.

  'C' programming language supports following control flow structures.

- **Types of Control Flow Structure**
  - **Sequential Control Structure**
  - **Selection or Decision Control Structure**
  - **Repetitive/Iterative/Loop Control Structure**
  - **Jump Statement**

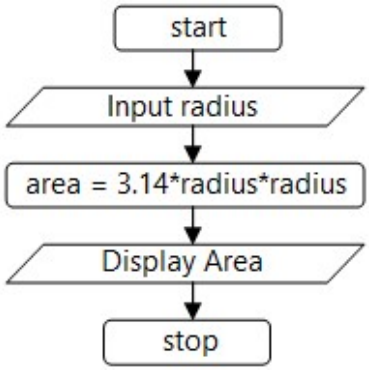
- **Sequential Control Structure**

  A program is a set of instructions which are normally executed sequentially in the order in which they are written.

  Sequential control flow means step by step or line by line execution of all program statements. Sequential programming can also be called ***'Linear programming'***. The sequential programs are non-modular in nature. That is, reusability of code is not possible. Thus they are difficult to maintain and understand.

  Consider an example of finding the area of circle. The area of the circle is found out using the formula ***pi*r**r***, where ***'r'*** is the radius of the circle and ***'pi'*** is the constant.

  The sequential flow structure can be pictorially resented as bellow.



Thus, in this case the objective is simple. That is, the task which is to be performed includes:

- Getting the input from the user(radius)
- Manipulating the required result and
- Printing the result

Hence, one can understand that the objective is clear and therefore, the program will have statements that are placed sequentially and there is no decision involved in this process.

Program :

```
/* to demonstate sequential control structure*/
#include<stdio.h>
void main();
{
float radius,area,pi=3.14;
clrscr();
printf("Enter radius of circle:");
scanf("%f",&radius);
area=pi*radius*radius;
printf("\n Area of circle:%f",area);
getch();
}
```

Output:

```
Enter radius of circle:2
Area of circle:12.56
```

- **Selection or Decision Control Structure**

A program is usually not limited to a linear sequence of instructions. In real life, a program usually needs to change the sequence of execution according to some conditions. Selective control structure modifies usual sequential flow of program.

In 'C' there are many control structures that are used to handle conditions and the resultant decisions.

Conditional operator(? :) is also the example of selection control structure, but in general there are following types of ***Selection/Decision Control Structure*** in ***'C'***.

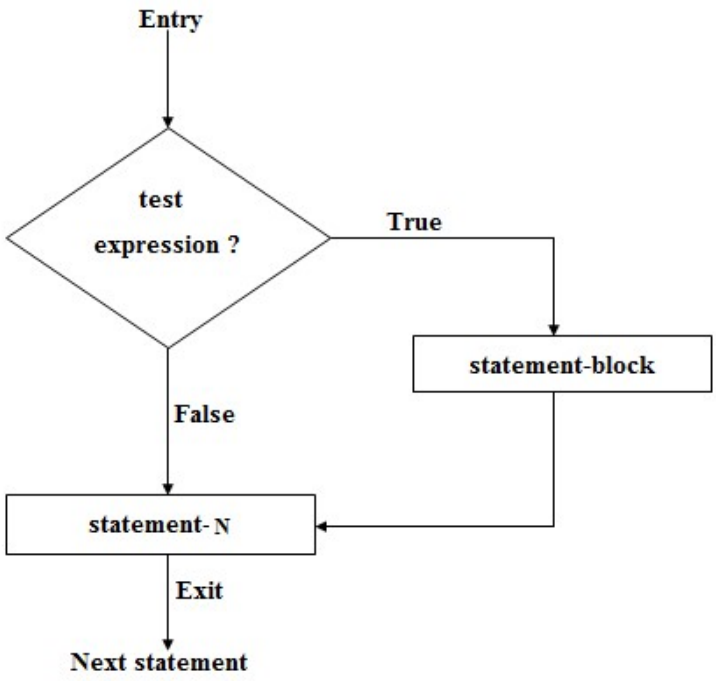
1) **Simple if Statement**
2) **if....else Statement**
3) **nested if....else Statement**
4) **if....else ladder Statement**
5) **switch Statement**

1) Simple if Statement

The if statement is powerful decision making statement and is used to control the flow of execution of statement. It allows the computer to evaluate the expression first and then depending on whether the value of the expression (relation or condition) is 'TRUE(1)' or 'FALSE(0)' , it transfer the control to a particular statement.

This point of program has two path to follow, one for TRUE condition and other for FALSE condition.

Following diagram shows it's working.



Syntax:

```
If(test expression)
{
        Statement-block;
}
        Statement-N;
```

The statement block may be a single statement or a group of statements. If the test expression/condition is true, the statement block and statement –N both will get executed; otherwise statement block will be skipped and the execution will jump to the statement-N.

Following program demonstrate the simple if statement when test expression/condition gets false.

Program -1:

```
/* simple if statement when condition gets false*/
#include<stdio.h>
#include<conio.h>
void main();
{
int a=4,b=7;
clrscr();
if(b<a)
{
     printf("Welcome...!");
     printf("\n SPIRIT AND SPARK GROUP");
}
     printf("\n TITWALA EAST,THANE");
getch();
}
```

Output:

```
TITWALA EAST,THANE
```

Program-2 :

```
/* simple if statement when condition gets true*/
#include<stdio.h>
#include<conio.h>
void main();
{
int a=4,b=7;
clrscr();
if(b>a)
{
     printf("Welcome...!");
     printf("\n SPIRIT AND SPARK GROUP");
}
     printf("\n TITWALA EAST,THANE");
getch();
}
```

Output:

```
Welcome...!
SPIRIT AND SPARK GROUP
TITWALA EAST,THANE
```

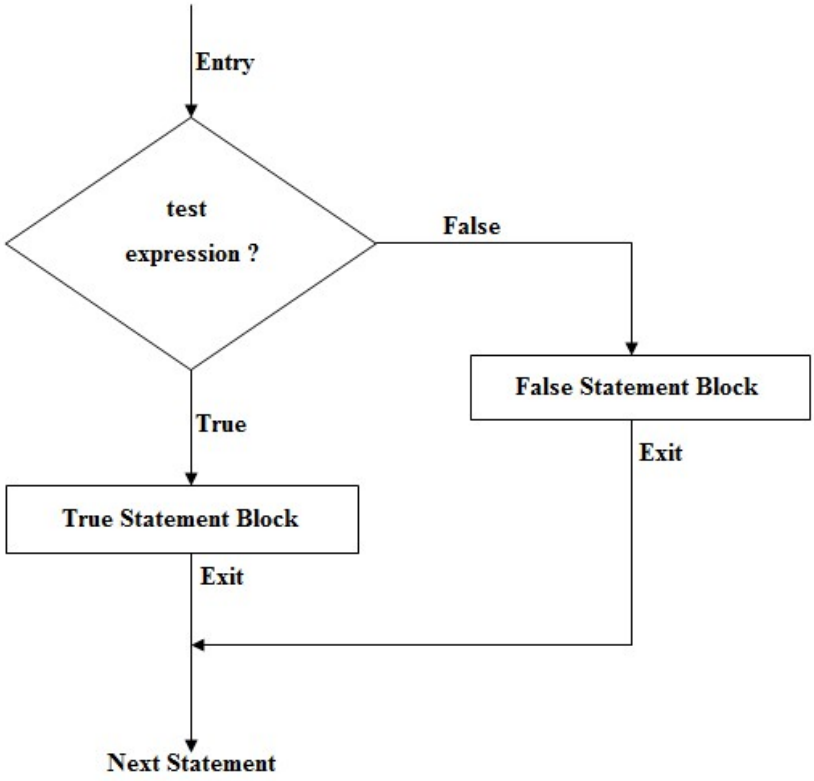
Note:

- *More than one condition can be written inside the if statement using logical operators.*
  e.g. if(a>b && a!=0)
- *Constructing body i.e. opening and closing braces are optional, but when if statement have multiple lines it is necessary to construct body.*

2) if....else Statement

The if....else statement is an extension of the simple if statement. If the expression/condition is true, then only true block statements will gets executed; otherwise false block statements executes only.

Following diagram shows it's working.



if....else statement has following three syntax , it can be used in program according to need.

Syntax-1:

```
If(test expression)
        True-Statement-Block;
else
        False-Statement-Block;
```

Program :

```
/* to demonstrate if....else statement with single line */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int a,b;
      clrscr();
      printf("Enter first number:");
      scanf("%d",&a);
      printf("Enter second number:");
      scanf("%d",&b);
      if(a>b)
            printf("\n first number is greater");
      else
            printf("\n second number is greater");
      getch();
      }
```

Output:

```
Enter first number:7
Enter second number:5
first number is greater
```

Syntax-2:

```
If(test expression)
{
      True-Statement-Block;
}
else
      False-Statement-Block;
      Statement-N;
```

Using above syntax , if test expression/condition is true then true statements block will be executed and also statement-N will be executes. It only skips the first statement/line to execute i.e. false statement block.

If test expression/condition becomes false then only false statement block and statement-N will be executed.

Program :

```
/* to demonstrate if....else statement with single line */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int a,b;
      clrscr();
      printf("Enter value for a:");
      scanf("%d",&a);
      printf("Enter value for b:");
      scanf("%d",&b);
      if(a>b)
      {
            printf("\n a is greater than b");
      }
      else
            printf("\n b is greater than a");
            printf("\n this is comparison  between two numbers
      ")

      getch();
      }
```

When we provide input as a= 7 and b=5 then it will shows following output.

Output:

```
Enter value for a:7
Enter value for b:5
a is greater than b
this is comparison between two numbers
```

If we provide input as a= 5 and b=7 then it will shows following output.

Output:

```
Enter value for a:5
Enter value for b:7
b is greater than a
this is comparison between two numbers
```

Syntax-3:

```
If(test expression)
{
        True-Statement-Block;
}
else
{
        False-Statement-Block;
}
        Statement-N;
```

Using above syntax , if test expression/condition is true; it will be executes all statements of true statement block body and also executes statement-N. False statement block will be skipped.

If test expression/condition becomes false, it will be executes all statements of false statement block body and also executes statement-N. True statement block will be skipped.

Program :

```
/* to demonstrate if....else statement with statement body */
        #include<stdio.h>
        #include<conio.h>
        void main();
        {
        int a,b;
        clrscr();
        printf("Enter value for a:");
        scanf("%d",&a);
        printf("Enter value for b:");
        scanf("%d",&b);
        if(a>b)
        {
              printf("\n a is greater than b");
              printf("\n b is smallest");
        }
        else
        {
              printf("\n b is greater than a");
              printf("\n a is smallest");
```

```
}
printf("\n this is comparison  between two numbers ");
getch();
}
```

Output:

```
Enter value for a:7
Enter value for b:5
a is greater than b
b is smallest
this is comparison between two numbers
```

3) nested if....else Statement

Sometimes we need t check/test for multiple decision depending on one another. This can be accomplished by using nested conditions.

When conditions are nested the if....else statement block may contain another if....else statement block within themselves.

In nesting programmer must be careful to keep track of different ***if's*** and corresponding ***else's.***

Programmer can use more than one if....else statement in nested form as follows.

1) ***Example-1:***

```
if(text expression-1)
{
    if(text expression-2)
    {
    Statement-1;
     }
    else
    {
    Statement-2;
    }
}
else
{
Statement-3
}
Statement-N
```

In above nested if....else format, if test expression-1 becomes false then the statement-3 and statement-N will be executed; Otherwise it will continue to perform the second test expression.

If the test expression-2 is true, then statement-1 will be executes otherwise statement-2 will be evaluated and then control will be transferred to the statement-N.

2) ***Example-2:***

```
if(text expression-1)
{
    Statement-1
}
else
{
if(text expression-2)
    {
    Statement-2;
     }
    else
    {
    Statement-3;
    }
}
Statement-N
```

In above nested if....else format, if test expression-1 becomes true then statement-1 will be executed and then control will be transferred to the statement-N.

Otherwise test expression-1 will be false, then control will be moves to the else block. At this stage test expressin-2 will be tested, if is true; then statement-2 will be executed otherwise statement-3 will be evaluated and control will be transferred towards the statement-N.

3) ***Example-3:***

```
if(text expression-1)
{
    if(text expression-2)
    {
    Statement-1;
     }
    else
    {
```

```
    Statement-2;
    }
}
else
{
if(text expression-3)
    {
    Statement-3;
     }
    else
    {
    Statement-4;
    }
}
Statement-N
```

In above nested if....else format, if test expression-1 becomes true then test expression-2 will be tested, if test expression-2 is true, then statement-1 will be executed. Otherwise statement-2 will be executed and control will be moves to evaluate the statement-N.

If test expression-1 becomes false then control will moves toward the else block, at this stage test expression-3 will be tested; if it is true, then statement-3 will be executed otherwise statement-4 will be executed and control will moves toward the statement-N.

Program :

```
/* to demonstrate nested if....else statement */
          #include<stdio.h>
          #include<conio.h>
          void main();
          {
          int num;
          clrscr();
          printf("Enter number:");
          scanf("%d",&num);
          if(num >0)
          {
                if(num <10)
                {
                printf("\n Entered number is greater than zero and
          less than 10");
```

```
        }
        else
        {
        printf("\n Entered number is greater than zero and
greater than 10 also");
        }
}
else
{
        if(num ==0)
        {
        printf("\n Entered number is zero");
        }
        else
        {
        printf("Entered number is less than zero");
        }
}
printf("\n this is demo of Nested if…else statement ");
getch();
}
```

Output:

```
Enter number:0
Entered number is zero
this is demo of Nested if…else statement
```

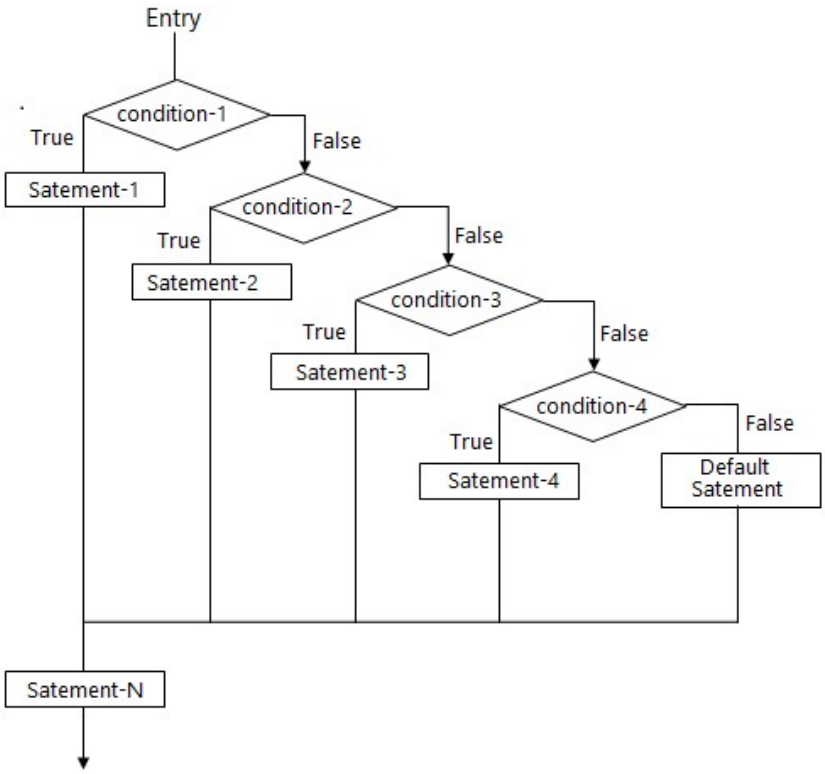
4) if....else ladder Statement

There is another way of putting ***if*** together when multipath decisions are involved. A multipath decision is a chain of ***if*** in which the statement associated with each else is an ***if*** .

Each condition/test expression is evaluated in order or particular sequence and if any condition/test expression is true, the corresponding statement is executed and the remaining of the chain is terminated.

The final else statement is only executed none of the previous condition satisfied.

Following diagrams gives the idea about the flow of if....else ladder statement.



Syntax:

```
 if(test expression-1)
{
        Statement-1;
}
else if(test expression-2)
{
        Statement-2;
}
------------------------
------------------------
------------------------
else if(test expression-n)
{
        Statement-n;
}
else
Statement-N;
```

Program :

```
/* to demonstrate if....else ladder statement */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int num;
      clrscr();
      printf("Enter number:");
      scanf("%d",&num);
      if(num >0)
      {
            printf("\n Entered number is positive");
      }
      else if(num <0)
      {
            printf("\n Entered number is negative");
      }
      else
            printf("\n Entered number is zero");
      getch();
      }
```

Output:

```
Enter Number:7
Entered number is positive
```

5) Switch Statement

Switch statement is a powerful statement used to handle many alternatives. Switch is a keyword by using switch we can create a selection statement with multiple choices. Multiple choices can be constructed by using case keyword.

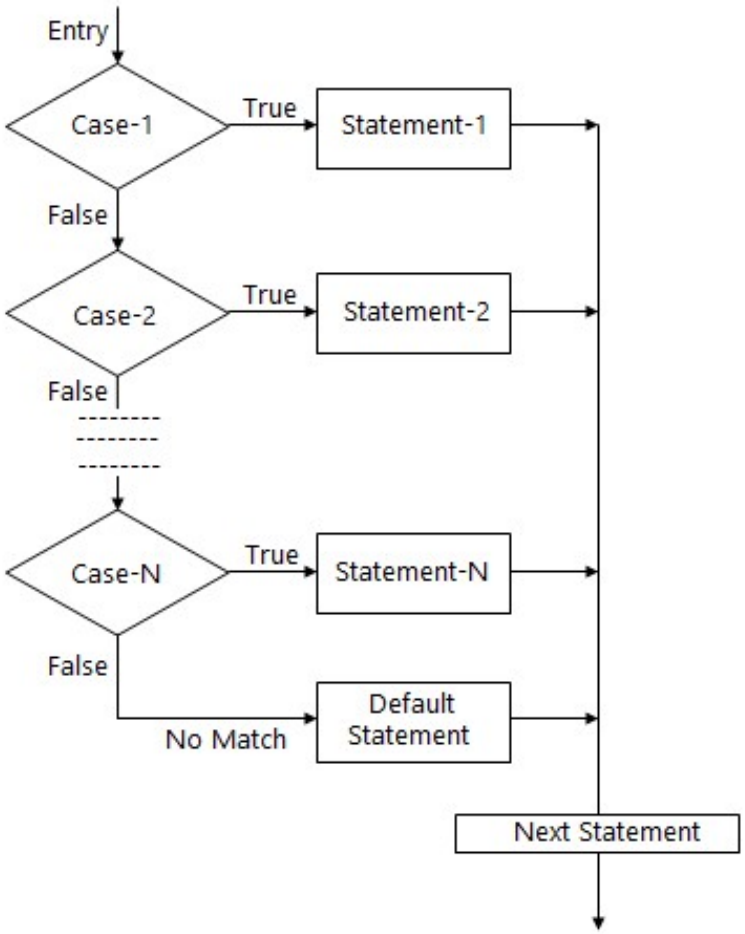
When we working with switch it requires a condition or an expression of integral type i.e. int, long, short or char etc. Case keyword requires constant expression or constant type integer only.

The following rules must be applied to construct switch.

- The expression used in a switch statement must have an integral or enumerated type i.e. int, long, short or char etc.
- You can have any number of case statements within a switch case is followed by the unique value with colon.
- The case followed by the value must be the same data type as the variable in the switch, and it must be a constant or literal.

- When the variable being switched and equal to the particular case, the statement following that case will execute until 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.
- A switch statement can have an optional default case can be used for performing a task when none of the cases is true. The break keyword is optional in default case.

Following flowchart shows the working of switch statement.



Syntax:

```
switch(test expression)
{
   case 1:
        Statement-1;
        break;
   case 2:
        Statement-2;
        break;
        ---------------
        ---------------
   case N:
        Statement-N;
        break;
   default:
        default Statement;
        break;
}
```

Program :

```
/* to demonstrate the switch statement */
     #include<stdio.h>
     #include<conio.h>
     void main();
     {
     int day;
     clrscr();
     printf("Enter the number to select day:");
     scanf("%d",&day);
     switch(day)
     {
     case 1:
          printf("You select Sunday");
          break;
      case 2:
          printf("You select Monday");
          break;
     case 3:
          printf("You select Tueday");
          break;
```

```
case 4:
        printf("You select Wednessday");
        break;
case 5:
        printf("You select Thursday");
        break;
case 6:
        printf("You select Friday");
        break;
case 7:
        printf("You select Saturday");
        break;
default:
printf("Day not found for this selection");break;
}
getch();
}
```

Output:

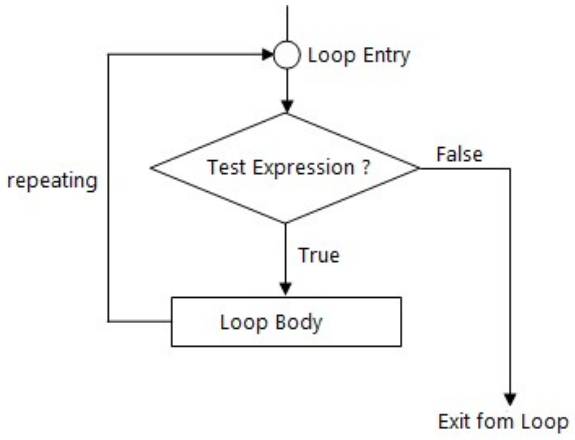
```
Enter the number to select day:7
You select Saturday
```

- **Repetitive/Iterative/Loop Control Structure**

The statement which can be executed for a specified set of times until the given condition satisfy is known as repetitive control structure. It is also known as iterative or loop control structure.

In programming language loops are used to execute a set of statements repeatedly until a particular condition meets.

Following diagram shows the working of loop.



Sequence of statement is executed until a specified condition is true. This sequence of statements to be executed is kept inside the curly braces{ }, it is called as loop body, after every execution of loop body condition is checked and if it is found to be true the loop body is executed again; when condition becomes false control will stop the execution of loop body and exit from the loop.

There are three types of loops in 'C' language.

1) **while loop**
2) **do....while loop**
3) **for loop**

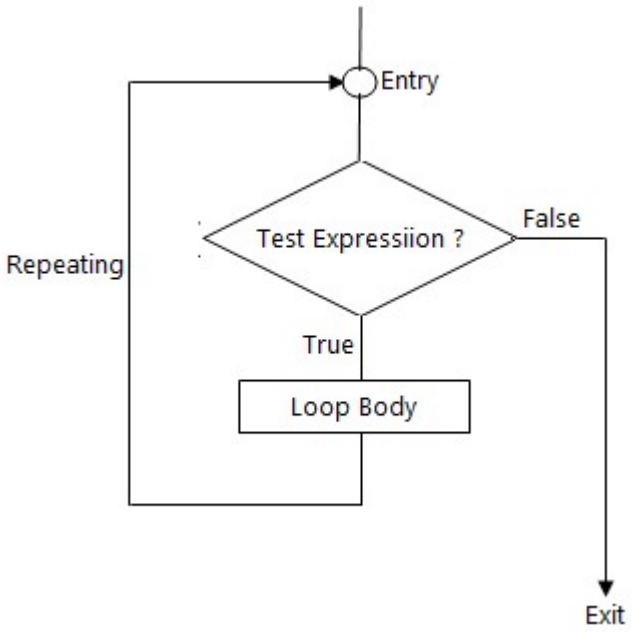
1) while loop

The while loop keeps repeating an action until an test expression/condition becomes false. This is useful where the programmer does not know in advance how many times the loop will be traversal/repeats.

While loop contains test expressions/conditions in parentheses in front of while keyword, through which loop body is controlled. It is also known as entry control loop.

The loop continues as long as the test expression is true. It will stop when the test expression becomes false. Programmer need to make sure that the expression will stop at some point otherwise it will become an infinite loop.

Following flowchart shows working of while loop.



Syntax:

```
while(test expression)
{
        Statement-1;
        ---------------
        ---------------
        Statement-N;
}
```

Program1 :

```
/* program to demonstrate the while loop */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int a=1;
      clrscr();
            while(a<11)
            {
            printf("\n%d",a);
            a++;
            }
      getch();
      }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Program2 :

```
/* printing table of 2 using while loop */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int i=1;
      clrscr();
      while(i<=20)
      {
            if(i%2==0)
            {
            printf("\n%d",i);
            }
            i++;
      }
      getch();
      }
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

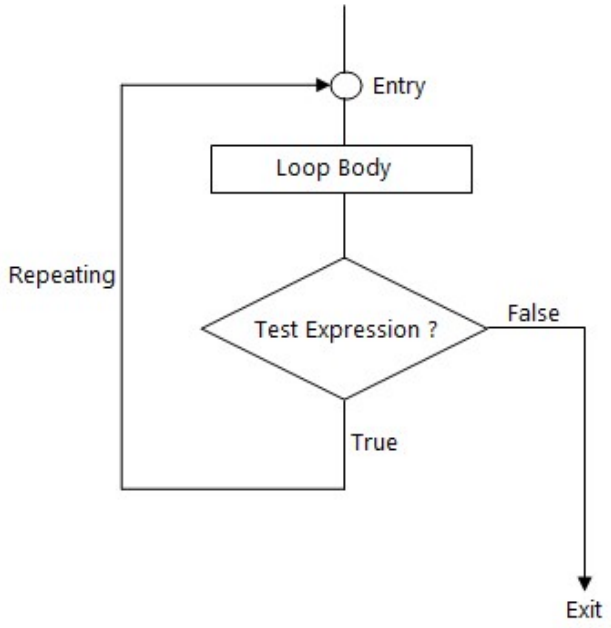
2) do....while loop

This is very similar to the while loop except that the test expression occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. This type of loop is called as exit control loop.

In this loop, code inside body of do is executed first after that test expression is tested. If it is true, then code inside body of do is executed and the process continues until test expression becomes false.

Notice that, there is semicolon at the end of while( ); in do....while loop.

Following flowchart shows the working of do....while loop.



Syntax:

```
do
{
        Statement-1;
        ---------------
        ---------------
        Statement-N;
} while(test expression);
```

Program1 :

```
/* program to demonstrate the do....while loop */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int a=1;
      clrscr();
            do
            {
            printf("\n%d",a);
            a++;
            }while(a<6);
      getch();
      }
```

Output:

```
1
2
3
4
5
```

Program2 :

```
/* printing table of 2 using do....while loop */
    #include<stdio.h>
    #include<conio.h>
    void main();
    {
    int i=1;
    clrscr();
    do
    {
        if(i%2==0)
        {
        printf("\n%d",i);
        }
        i++;
    }while(i<=20);
    getch();
    }
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

3) for loop

The for loop works well where the number of iterations of the loop is known before the loop before the loop is entered. So we can say that, for loop is designed to perform a repetitive action for a pre-defined number of times.

It has following syntax.

Syntax:

```
for(initialization; test expression; increment/decrement)
{
        Statement-1;
        ---------------
        ---------------
        Statement-N;
}
```

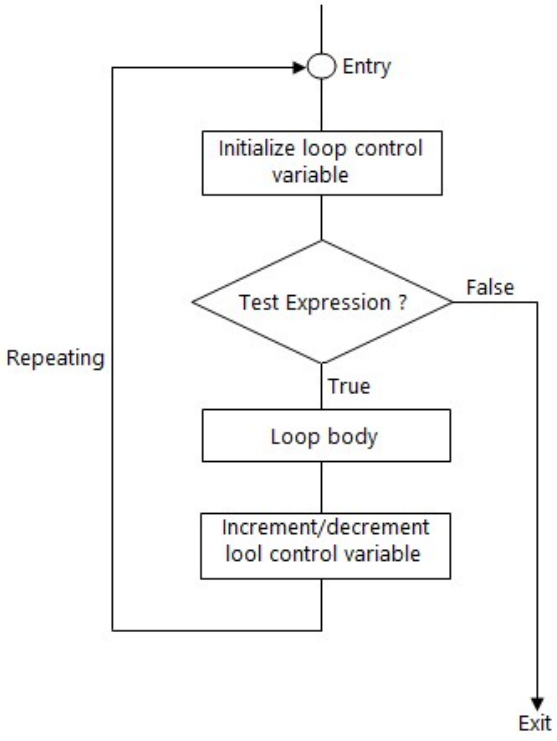
For loop consist of three parts i.e. initialization, test expression and third part is increment/decrement, which is separated by semicolon.

Initialization is run before the loop is entered. The purpose of initialization is to begin /initialize the loop control variable.

Second part is ***test expression***; if the test expression is false, then for loop is terminated. But, if test expression is true then the codes within loop body will be executed.

Third part is ***increment/decrement*** the loop control variable. It is used to increment or decrement the loop control variable according to condition specified in loop variable. If increment/decrement is not specified according to test expression, it will cause errors or turns loop into infinity.

Following flowchart shows the working of do....while loop.



The working of for loop is very simple. When it encountered the loop control variable will be initialized, after this test expression is checked. If the test expression becomes false, the loop is terminated; otherwise the code within loop body will be executed. After executing loop body, the loop control variable is incremented or decremented according to the test expression. Again test expression is tested and the same process is continued until test expression becomes false.

Program:

```
/* program to demonstrate the for loop */
      #include<stdio.h>
      #include<conio.h>
      void main();
      {
      int count;
      clrscr();
            for(count=1;count<=5;count++)
            {
            printf("\n%d",count);
            }
      getch();
      }
```

Output:

```
1
2
3
4
5
```

- **Jump Statements**

'C' supports following keywords as 'Jump Statements' which can be used to jump at a specific position in program.

- break
- continue
- goto
- return

1) break statement

The break statement is used to forcefully terminate loops or to exit from a switch. It can be used within a while loop, do….while loop, for loop or a switch statement.

Sometimes, it is necessary to exit immediately from the loop as soon as specified condition/test expression is satisfied. When break statement is used inside a loop, then it can cause to terminate from a loop. The statements after break statements are skipped.

When and where break statement is used in program?

- Using break is always optional but it should be place within the loop body or switch body.
- In implementation where we know the maximum number of repetitions but some condition is there, where we need to terminate the repetition process, then break statement is used.

Syntax:

```
any loop/switch(test expression)
{
        Statement-1;
        ---------------
        break;
        ---------------
        Statement-N;
}
```

Program1 :

```
/* program to demonstrate break statement */
      #include<stdio.h>
      #include<conio.h>
      int main();
      {
      int i;
      clrscr();
            for(i=0;i<=10;i++)
            {
            if(i==5)
            {
                  printf("Exit from loop when i=5");
                  break;
            }
            printf("%d\n",i);
            }
      getch();
      }
```

Output:

```
0
1
2
3
4
Exit from loop when i=5
```

2) Continue statement

Sometimes, it is required to skip a part of a body of loop under specific condition. So that to meet this requirement 'C' supports 'continue' statement.

The working of continue statement is much similar to the break statement but difference is that it can not terminate the loop. It causes the loop to be continued with next iteration after statement in between specified condition. Continue statement simply skips statements and continues next iteration.

When and where continue statement is used in program?

- Using continue is always optional but it should be placed within the loop body only.
- In implementation where we know the maximum number of repetitions but it has some test expression/condition, where we need to skip the statements from repetition process and after skipping it continues the next statements.

Syntax:

```
any loop(test expression)
{
        Statement-1;
        ---------------
        continue;
        ---------------
        Statement-N;
}
```

Program1 :

```
/* program to demonstrate continue statement */
      #include<stdio.h>
      #include<conio.h>
      int main();
      {
      int i;
```

```
clrscr();
      for(i=0;i<10;i++)
      {
      if(i==5 || i==6)
      {
            printf("skiping % to display by using
      continue statement\n",i);
            continue;
      }
      printf("%d\n",i);
      }
getch();
}
```

Output:

```
0
1
2
3
4
skiping 5 to display by using continue statement
skiping 6 to display by using continue statement
7
8
9
```

3) goto statement

The goto statement requires a label for it's operation/implementation. A label is valid identifier followed by a colon.

It is used to directly jump at any specified label in a function, it should be strictly follows that, the label must be in same function in which 'goto' statement is used. Through the goto statement you can not jump between any other functions.

Syntax:

```
goto label;                    label:
Statement-1;                   Statement-1;
----------          OR         ----------
----------                     ----------
label:                         goto label;
Statement-N;                   Statement-N;
```

Program1 :

```
/* program to demonstrate continue statement */
    #include<stdio.h>
    #include<conio.h>
    int main();
    {
    int a,b;
    clrscr();
    printf("Enter first number:");
    scanf("%d",&a);
    printf("\n Enter second number:");
    scanf("%d",&b);
        if(a>b)
        {
            goto First;
        }
        else
        {
            Goto Second;
        }
    First:
    printf("First number is greater than second number\n");
    goto Exit;

    Second:
    printf("second number is greater than first number\n");
    goto Exit;

    Exit:
    printf("This is comparision between two numbers");
    getch();
    }
```

Output:

```
Enter first number:4
Enter second number:7
second number is greater than first number
This is comparision between two numbers
```

4) return statement

The return statement we will discuss in ***chapter-10, i.e. Functions***.

7. ARRAYS

- **Introduction**

  Variable is a memory location where user stores single value at a time. It means in case of simple variable only single value can be represented at a time by single identifier. e.g. if we would like to store the details of the entire students in single list, we can not store these data using simple variable through single identifier.

  For this purpose 'C' has provided the derived data type ***Array***. The basic data type of array is formed using ***fundamental data types*** only. An array in 'C' programming is the number of memory locations, in each of those we can store the same data type and the value of all those memory location can be access by using same identifier/variable name.

- **What is an *Array?***

  Array can be defined as, '***an array is collection of variables having same name and same data type*** *'.* Array provides ability to use a single name to represent a collection of items. Individual array elements are identified by an integer index called as ***subscript***. In 'C' programming index begins from zero and it is always written in square brackets.

  - ***Declaration of Array***

    An array should be declared before it is used in 'C' program. The array declaration tells the compiler.

    - The type of the array
    - The name of the array
    - The number of elements in the array or the size of array.

    To declare an array, use the following syntax.

    <type_of_array> <name_of_array> <size_of_array>

    - **Type of array**

      It is the data types of an elements that an array stores. If array stores integer elements then type of array is 'int'. If array stores character elements then the type of array is 'char'. If type of elements in array is structure objects then type of array becomes the structure.

      This array of structure we will discuss in ***chapter-12, i.e. Structure and Unions.***

- **Name of array**

  This is an identifier as name given to the array. It can be any string except keyword of 'C', but it is usually suggested that some standard must be followed while naming array. Atleast name should be in context with what is being stored in array.

- **Size of array**

  This is the value in subscript/index[ ], indicates the number of elements that array stores.

- ***Initialization of Array***

  An array can be initialized in the following way.

  First way is to initializing the each element separately.

  e.g.

```
int arr[5];
arr[0]=1;
arr[1]=2;
arr[2]=3;
arr[3]=4;
arr[4]=5;
```

  And another way is to initializing the array at the time of declaration.

  e.g.

```
int arr[5]={1,2,3,4,5};
```

Note :

- *If array has size 'N' and you want to access N^{th} element, then compiler always writes N- 1^{th} element. This is caused due to the "off-by-one" errors.*
- *If we does not specified or initialized the elements in the array, it will collect the garbage value in it.*
- *If size of array is large than subscript then compiler will writes the remaining element as zero.*
- *If elements/values are specified for the array then it is not necessary to mention the array.*

  ***e.g. int arr[ ]={1,2,3,4,5}**; is equivalent to **int arr[5]={1,2,3,4,5}**;*
- *If size of array is too small as that of elements will be stored in the array, then compiler will flash an error as **'Too many to initializers'** .*
- *To processing an array loop is required.*

Program1 :

```
/* to demonstate initialization of array with
initializing each element separately */
#include<stdio.h>
int main();
{
int arr[];
int i=0;
clrscr();
for(i=0;i<5;i++)
{
arr[i]=i;
printf("array index values are :%d\n",i);
}
getch();
return 0;
}
```

Output:

```
0
1
2
3
4
```

Program2:

```
/* to demonstate initialization of array at the time of
declaration */
#include<stdio.h>
int main();
{
int arr[5]={5,7,8,9,6};
int i=0;
clrscr();
for(i=0;i<5;i++)
{
printf("array element at index %d is %d \n",i,arr[i]);
}
getch();
return 0;
}
```

Output:

```
array element at index 0 is 6
array element at index 1 is 7
array element at index 2 is 8
array element at index 3 is 9
array element at index 4 is 6
```

Note :

- *Initializing array at the time of declaration is also called as compile time array initialization.*
- *Initializing array using scanf() function ,it means accepting values in array through users is called as runtime array initialization.*

Program3:

```
/* program to implement runtime array or array
accepting elements from user  */
#include<stdio.h>
void main();
{
int arr[5],i;
clrscr();
for(i=0;i<=4;i++)//loop 'i' in range 0 to 4
{
printf("Insert an Element into Array:\n");
scanf("%d",&arr[i]);//read value in arr[i]location
}
printf("\n Element in Array are:\n");
for(i=0;i<=4;i++)
{
printf("array element at index %d is %d \n",i,arr[i]);
//print number from arr[i]
}
getch();
}
```

Output:

```
Insert an Element into Array:10
Insert an Element into Array:30
Insert an Element into Array:20
Insert an Element into Array:25
Insert an Element into Array:43
```

```
Element in Array are:
array element at index 0 is 10
array element at index 1 is 30
array element at index 2 is 20
array element at index 3 is 25
array element at index 4 is 43
```

- **Types of Array**

  'C' support following types of arrays:

  - **Single /one dimensional Array**
  - **Two dimensional Array**
  - **Multidimensional Array**

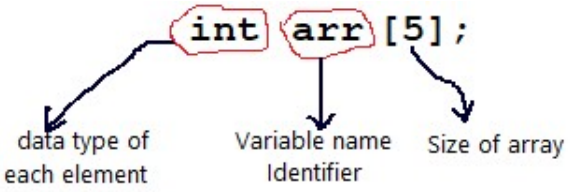
- **Single /one dimensional Array**

  An array having only one subscript/index is known as single /linear or one dimensional array. Single dimensional array is a list of data items(variables) that is given a variable name using only one subscript/index.

  It has following syntax.

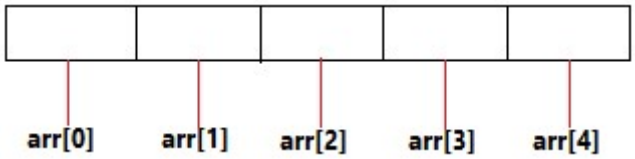
  <type_of_array> <name_of_array> <size_of_array>

  This syntax we had already discussed in previous section i.e. ***Declaration of Array.*** e.g. if we want to represent a set of five number by an array variable, we can declare the array as follows.



  Above declaration will declare a single dimensional array having five integer variables. One thing to be notice that, the index of array start with zero. The system reserves five storage locations of memory.

  Following diagram shows it's conceptual view.



Program:

```
/* Find sum of array elements*/
#include<stdio.h>
void main();
{
int arr[5],i,sum=0;
clrscr();
for(i=0;i<=4;i++)//loop 'i' in range 0 to 4
{
printf("\n Insert an Element into Array:");
scanf("%d",&arr[i]);//read value in arr[i]location
sum=sum+arr[i];
}
printf("\n sum of array element is:%d",sum);
getch();
}
```

Output:

```
Insert an Element into Array:8
Insert an Element into Array:9
Insert an Element into Array:5
Insert an Element into Array:9
Insert an Element into Array:14
sum of array element is:45
```

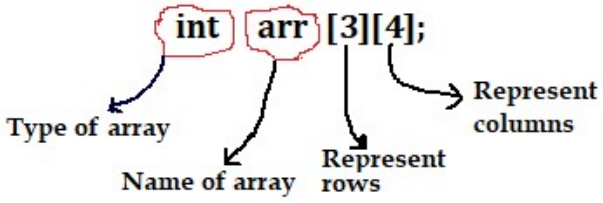
- **Two dimensional Array**

If an array having exactly two subscript/index, then it is called as two dimensional array. It will store the values in row and column format. e.g. in the matrix form.

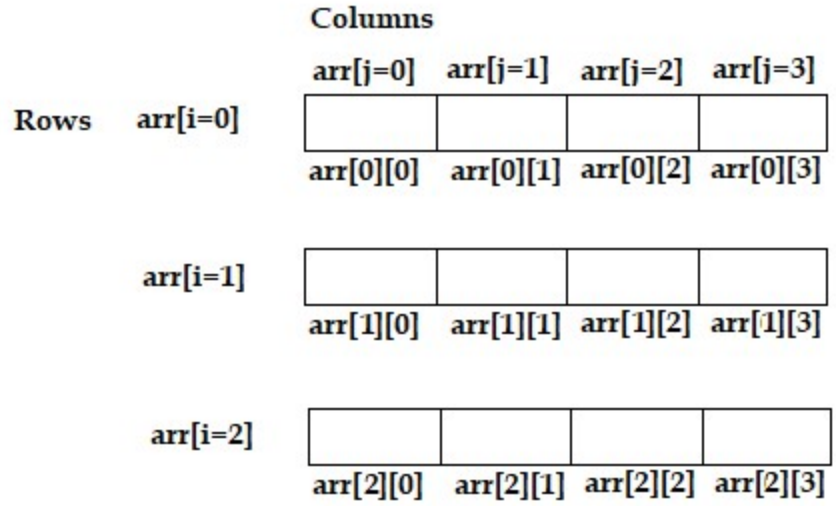
It has following syntax.

<type_of_array> <name_of_array> <row_size ><column_size>

This syntax we had already discussed in previous section, just *< size_of_array >* is specified with two subscript *i.e. rows and columns.* e.g. if we want to represent an array with 3 rows and 4 columns, it can be declared using 2-D array as follows.



Following diagram shows it's conceptual view.



The above array ***arr*** contains three rows and four columns, so it is said to be 3-by-4 array. Every element in array ***arr*** is identified by an element name of the form ***arr[i][j]; arr*** is the name of array and ***i,j*** are the subscript/index that uniquely identify each element in array ***arr*** . Notice that, all element in the first row have the first value of subscript/index i.e. zero(0) and the all elements in the third column have the third subscript/index i.e. 2.

Program:

```
/* program to demonstrate 2D array */
#include<stdio.h>
#include<conio.h>
int main();
{
int arr[5][2]={{1,2},{3,4},{5,6},{8,7},{16,25}}
int i,j;
clrscr();
for(i=0;i<5;i++)//represent rows
{
printf("\n");
      for(j=0;j<2;j++)//represent columns
      {
      printf("\t arr[%d][%d]=%d",i,j,arr[i][j]);
      }
}
getch();
return 0;
}
```

Output:

```
arr[0][0]=1          arr[0][1]=2
arr[1][0]=3          arr[1][1]=4
arr[2][0]=5          arr[2][1]=6
arr[3][0]=8          arr[3][1]=7
arr[4][0]=16         arr[4][1]=25
```

- **Multidimensional Array**

An array whose elements are specified by more than tow subscript/index is referred as multi-dimensional array.

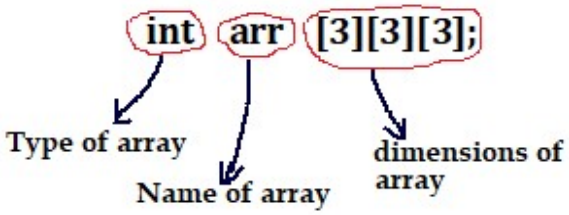
In 'C' programming an array can have two, three, four or more dimensions. More dimensions inn array means more data it can hold. It is difficult to manage and understand.

It has following syntax.

<type_of_array> <name_of_array> <[d1][d2][d3].....[dn] >

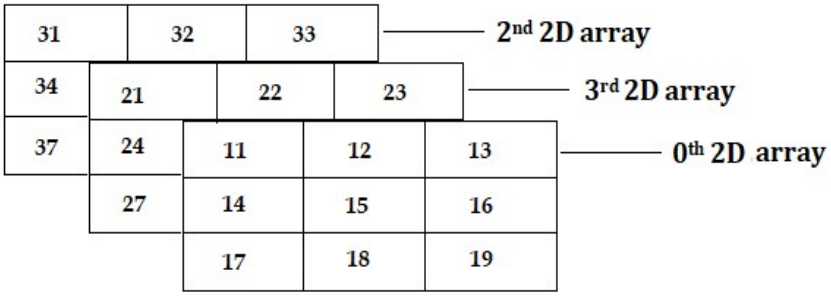
This syntax we had already discussed in previous section, just *< size_of_array >* is specified with more subscript/index i.e. *d1, d2, d3 ,.....dn.*

Where ***dn*** is the size of last dimension. For example, if we want to represent an array with three dimensions it can be declared as follows.

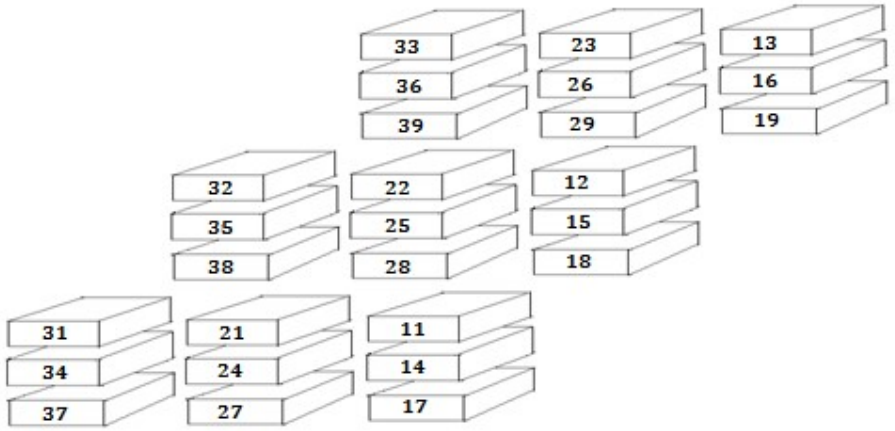


A multidimensional array can be assumed as an array of arrays of arrays, it means it is array(collection) of 2Dimensional arrays.

Following diagram shows the conceptual view of above 3D array.



To understand in depth it's conceptual view can be designed as follows.



Program:

```
/* program to demonstrate 3D array */
#include<stdio.h>
#include<conio.h>
int main();
{
int arr[3][3][3]={
{{11,12,13},{14,15,16},{17,18,19}},
{{21,22,23},{24,25,26},{27,28,29}},
{{31,32,33},{34,35,36},{37,38,39}}
};
int i,j,k;
clrscr();
printf("\n\t\t:::3D array elements are:::\n");
for(i=0;i<3;i++)
{
printf("\n");
   for(j=0;j<3;j++)
   {
    printf("\n");
    for(k=0;k<3;k++)
    {
    printf("\t arr[%d][%d][%d]=%d",i,j,k,arr[i][j][k]);
    }
   }
}
getch();
return 0;
}
```

Output:

```
:::3D array elements are:::

arr[0][0][0]=11   arr[0][0][1]=12     arr[0][0][2]=13
arr[0][1][0]=14   arr[0][1][1]=15     arr[0][1][2]=16
arr[0][2][0]=17   arr[0][2][1]=18     arr[0][2][2]=19

arr[1][0][0]=21   arr[1][0][1]=22     arr[1][0][2]=23
arr[1][1][0]=24   arr[1][1][1]=25     arr[1][1][2]=26
arr[1][2][0]=27   arr[1][2][1]=28     arr[1][2][2]=29

arr[2][0][0]=31   arr[2][0][1]=32     arr[0][0][2]=33
arr[2][1][0]=34   arr[2][1][1]=35     arr[0][1][2]=36
arr[2][2][0]=37   arr[2][2][1]=38     arr[0][2][2]=39
```

8. STRINGS

- **Introduction**

The group of integers can be stored in an integer array, similarly a group of characters can be stored in 1-dimentional character type array. Each character within the group will be stored within the group will be stored as one element of the array, that means each character occupies one location in an array. The null character '\0' is put after the last character. It indicates the end of string.

- **What is the *String?***

In simple word we can say that, '***string is collection of characters*** *'.* When we write word or sentence, it is treated as a string.

e.g. **'WELCOME TO SPIRIT AND SPARK'** is called as string. To work with string remember the following points.

— String in 'C' is also called as character array, which is used to manipulate text such as word and sentences.
e.g. **'S','A','N','G','H','D','E','E','P'**

— String is enclosed within double quotes.
e.g. **"SANGHDEEP"**

— Each character occupies 1-byte of memory.
String always terminated with null character **'\0'**
e.g. **char name[10]={'T','I','T','W','A','L','A','\0'}**

— Null character is having **ASSCII** value **0** i.e. **ASSCII** value of **'\0'=0.**

— As string is nothing but character array, so it is possible to access individual characters through it's subscript/index.
e.g. **char name[10]="SANGHDEEP";**
so , we can access individual character through it's subscript as,

```
name[0]= 'S';
name[1]= 'A';
name[2]= 'N';
name[3]= 'G';
name[4]= 'H';
name[5]= 'D';
name[6]= 'E';
name[7]= 'E';
name[8]= 'P';
name[9]= '\0';
```

- ***Declaration of String***

String is useful whenever we accept name of person, address of the person or some descriptive information. We cannot use string as data type. Because 'C' does not support such data type, instead we use array of character to store string.

Following syntax is used to declare string.

<char> <string_variable_name> <size >

e.g.

```
char city[10];
char name[20];
char message[50];
```

Note :

- *String/character array variable name should be legal 'C' identifier.*
- *String variable must have size specified with square bracket.*
  e.g. If you declare string variable as, `char city[];` it will cause compile time error.
- *Do not use 'string' as data type, because 'C' does not support 'string' as data type.*
- *When you are using string for other purpose than accepting and printing data type then you must include following header file in your code.*

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

- ***Initialization of string***

Whenever we declare a string then it will contain garbage values inside it. We have to initialize string or character array before using it. '*The process of assigning some legal default data/value to string is called initialization of string*'.

There are following ways to initialize string in 'C' programming.

1) **Initializing string using un-sized array of character**
2) **Initializing string directly**
3) **Initializing string using character pointer**

1) Initializing string using un-sized array of character

In this approach, length(size) of character array is not specified while initializing it. Array length is not specified while initializing it. Array length is automatically calculated by compiler.

Individual characters are written inside single quotes and separated by comma to form a group/list of characters. Complete list is wrapped inside pair of curly braces.

In this approach we have need to put the null character at the ending of the string to terminate the character array. Using this approach we can declare character array /string as follows.

```
char name[]={'S','A','N','G','H','D','E','E','P','\0'};
```

2) Initializing string directly

In this method we are directly assigning string to variable by writing text in double quotes. In this type of initialization, we does not need to put the null character at the end of string. It is appended automatically by the compiler.

Following example shows the initialization of string using direct approach.

```
char name[]="SANGHDEEP";
```

3) Initializing string using character pointer

Another way of initializing string is to declare character variable of pointer type, so that it can hold the base address of string. Base address means address of first array element i.e. address of character array **name[0].**

Also in this type of initialization, we don't need to put null character at the end of string. It is appended automatically by the compiler.

Following example shows the initialization of character array/string using character pointer.

```
Char*name="SANGHDEEP";
```

'*' this symbol indicates pointer variable, we will studding more about pointer in ***chapter-9, i.e. Pointers.***

Program1:

```
/* program to display string and display value stored
at each location */
#include<stdio.h>
#include<conio.h>
void main();
{
char name[10]="SANGHDEEP";
int i;
clrscr();
printf("\n Name is:%s",name);//printing complete string
```

```
printf("\n value stored at each location  are:\n");

for(i=0;i<10;i++)
{
printf("name[%d]=%c\n",i,name[i]);/*printing string
character by character*/
}
getch();
}
```

Output:

```
Name is:SANGHDEEP
value stored at each location  are:
name[0]= S
name[1]= A
name[2]= N
name[3]= G
name[4]= H
name[5]= D
name[6]= E
name[7]= E
name[8]= P
name[9]=
```

Program2:

```
/* program to calculate the length of string entered by
user*/
#include<stdio.h>
#include<conio.h>
void main();
{
char str[10];
int i=0;
clrscr();
printf("\n Enter string:");
scanf("%s",str);//accepting string from user
while(str[i]!=0)//checking the end of string
{
i++;//calculating the length of string
}
printf("Length of string %s is:%d",str,i);
getch();
}
```

Output:

```
Enter string:SANGHDEEP
Length of string SANGHDEEP is:9
```

Program3:

```
/* program to print reverse of string */
#include<stdio.h>
#include<conio.h>
void main();
{
char str[10];
int i=0,j=1;
clrscr();
printf("\n Enter string:");
scanf("%s",str);//accepting string from user
while(str[i]!=0)//checking the end of string
{
i++;//calculating the length of string
}
printf("\n The reverse of string is:");
for(j=i;j>=0;j++)//reverse loop
{
printf("%c",str[j]);//print each reverse character
}
getch();
}
```

Output:

```
Enter string:GOD
The reverse of string is:DOG
```

Program4:

```
/* program to convert uppercase to lowercase and vice-
versa */
#include<stdio.h>
#include<conio.h>
void main();
{
char str[10];
int i=0,j=1;
clrscr();
```

```
printf("\n Enter string:");
scanf("%s",str);//accepting string from user
while(str[i]!=0)//checking the end of string
{
      if(str[i]>=97 && str[i]<=122)/*checking whether
      characters are in lowercase*/
      {
      Str[i]=str[i]-32;/*converting lowercase to
      uppercase*/
      }
      else if(str[i]>=65 && str[i]<=96)/*checking
      whether characters are in uppercase*/
      {
      Str[i]=str[i]+32;/*converting uppercase to
      lowercase*/
      }
i++;//calculating the length of string
}
printf("\n The converted string is:%s",str);
getch();
}
```

Output:

```
Enter string:PRAnali
The converted string is:praNALI
```

- **String related library functions**

In previous section we had seen programs to manipulate the string. Always it is not possible to manipulate string manually, this makes programming complex and large.

To overcome from these problems 'C' supports string handling functions in standard library "***string.h***". To use string related library functions, it is necessary to include "***string.h***" header file with pre-processor directives #include such as **`#include<string.h>`** otherwise it will generates errors at runtime.

'C' standard library supports following string related functions to manipulate stings.

- **gets () and puts ()**
- **strlen ()**
- **strcpy ()**
- **strcat ()**

- **strcmp()**
- **strlwr()**
- **strupr()**

- **gets() and puts()**

gets() function reads the string from standard input and store it into variable, whereas, **puts()** function writes the string to standard output.

We had already studied about it, in ***chapter-4, i.e. Standard Input And Output functions*** , please refer it.

Program:

```
/*accepting string from user using gets function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str[10];
clrscr();
printf("\n Enter string:");
gets(str);//accepting string from user
getch();
}
```

Output:

```
Enter string:vrushali
```

- **strlen()**

strlen() function is used to calculate the length of the given string. To use it, we also require one integer variable with character array to store the length of string.

It has following syntax.

<integer_variable_name>= <strlen(name_of_characterarray)>;

strlen() function counts the number of characters until the escape/blank or end character reached and returns the integer value.

Following program demonstrate the **strlen()** function.

Program:

```
/*calculate the length of string using strlen()
function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str[20];
int length;
clrscr();
printf("\n Enter string:");
scanf("%s",str);//accepting string from user
length=strlen(str);//calculate the length of string
printf("The length of %s is:%d",str,length);
getch();
}
```

Output:

```
Enter string:programming
The length of programming is:11
```

- **strcpy()**

strcpy() function is used to copy the contents/characters of one string into another string.

It has following syntax.

strcpy<(destination_characterarray_name,source_characterarray_name)>;

strcpy() function always copy the content of second parameter into the first parameter.

Program:

```
/*to copy the string using strcpy() function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str1[]="SPIRIT", str2[]="SPARK";
clrscr();
```

```
printf("Before copying first string is:%s and \n second
string is:%s\n",str1,str2);
strcpy(str1,str2);//coping string from str2 to str1
printf("After copying first string is:%s and \n second
string is:%s\n",str1,str2);
getch();
}
```

Output:

```
Before copying first string is:SPIRIT and
second string is:SPARK

After copying first string is:SPARK and
second string is:SPARK
```

- **strcat()**

strcat() function is used to concatenates(joins) two given strings. It always concatenates source string at the end of destination string.

It has following syntax.

strcat<(destination_characterarray_name,source_characterarray_name)>;

Example:

strcat(str1,str2);

This concatenates **str2** at the end of **str1.**

We know that, each string in 'C' is ended up with null character('\0'); while performing **strcat()** operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after **strcat()** operation.

Program:

```
/*concatenating strings using strcat() function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str1[]="SPIRIT", str2[]="SPARK";
clrscr();
printf("\n first string is:%s and second string
is:%s",str1,str2);
```

```
strcat(str1,str2);//concatenate str2 to str1
printf("\n After string concatenation:%s",str1);
getch();
}
```

Output:

```
first string is:SPIRIT and second string is:SPARK
After string concatenation:SPIRITSPARK
```

- **strcmp()**

strcmp() function in 'C', compares two given strings and returns zero if they are same. If the length of first string is less than the length of second string, it returns less than zero value. Else the length of first string is greater than the length of second string, it returns greater than zero value.

strcmp() function is case sensitive i.e. "A" and "a" are treated as different characters.

It has following syntax.

strcmp<(first_characterarray_name,second_characterarray_name)>;

Note that, **strcmp()** function always used with if statement.

Program:

```
/*string comparision using strcmp() function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str1[20],str2[20];
clrscr();
printf("\n Enter first string:");
gets(str1);
printf("\n Enter second string:");
gets(str2);
if(strcmp(str1,str2)==0)
{
printf("strings are equal:");
}
else
printf("strings are not equal:");
```

```
getch();
}
```

Output:

```
Enter first string:RUCHA
Enter first string:RUTIKA
strings are not equal
```

- **strlwr()**

strlwr() function converts a given string into lowercase.
It has following syntax.

strlwr<(characterarray_name)>;

Program:

```
/*converting given string to lower case using strlwr()
function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str[20];
clrscr();
printf("\n Enter string to convert in lower case:");
scanf("%s",str);
strlwr(str)
printf("\n The converted string is:%s",str);
getch();
}
```

Output:

```
Enter string to convert in lower case:SANGHDEEP
The converted string is:sanghdeep
```

- **strupr()**

strupr() function converts a given string into uppercase.
It has following syntax.

strupr<(fcharacterarray_name)>;

Program:

```
/*converting given string to upper case using strupr()
function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main();
{
char str[20];
clrscr();
printf("\n Enter string to convert in upper case:");
scanf("%s",str);
strupr(str)
printf("\n The converted string is:%s",str);
getch();
}
```

Output:

```
Enter string to convert in upper case:titwala
The converted string is:TITWALA
```