

### **Algorithm**

It is step-by-step procedure to solve any given problem. Persian scientist, astronomer and mathematician **Abdullah Muhammad bin Musa al-Khwarizmi**, often cited as “The father of Algebra”, was indirect responsible for the creation of the term “Algorithm”.

### **Characteristics of Algorithm**

- Precision – the steps are precisely stated (defined).
- Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness – the algorithm stops after a finite number of instructions are executed.
- Input – the algorithm receives input.
- Output – the algorithm produces output.
- Generality – the algorithm applies to a set of inputs.

### **Advantages of algorithm**

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program

### **Disadvantages of algorithm.**

1. Writing algorithm takes a long time.
2. An Algorithm is not a computer program, it is rather a concept of how a program should be.

**Write an algorithm to add two numbers entered by user.**

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Accept values num1 and num2.
Step 4: Compute sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

**Write an algorithm to find the largest among three different numbers entered by user.**

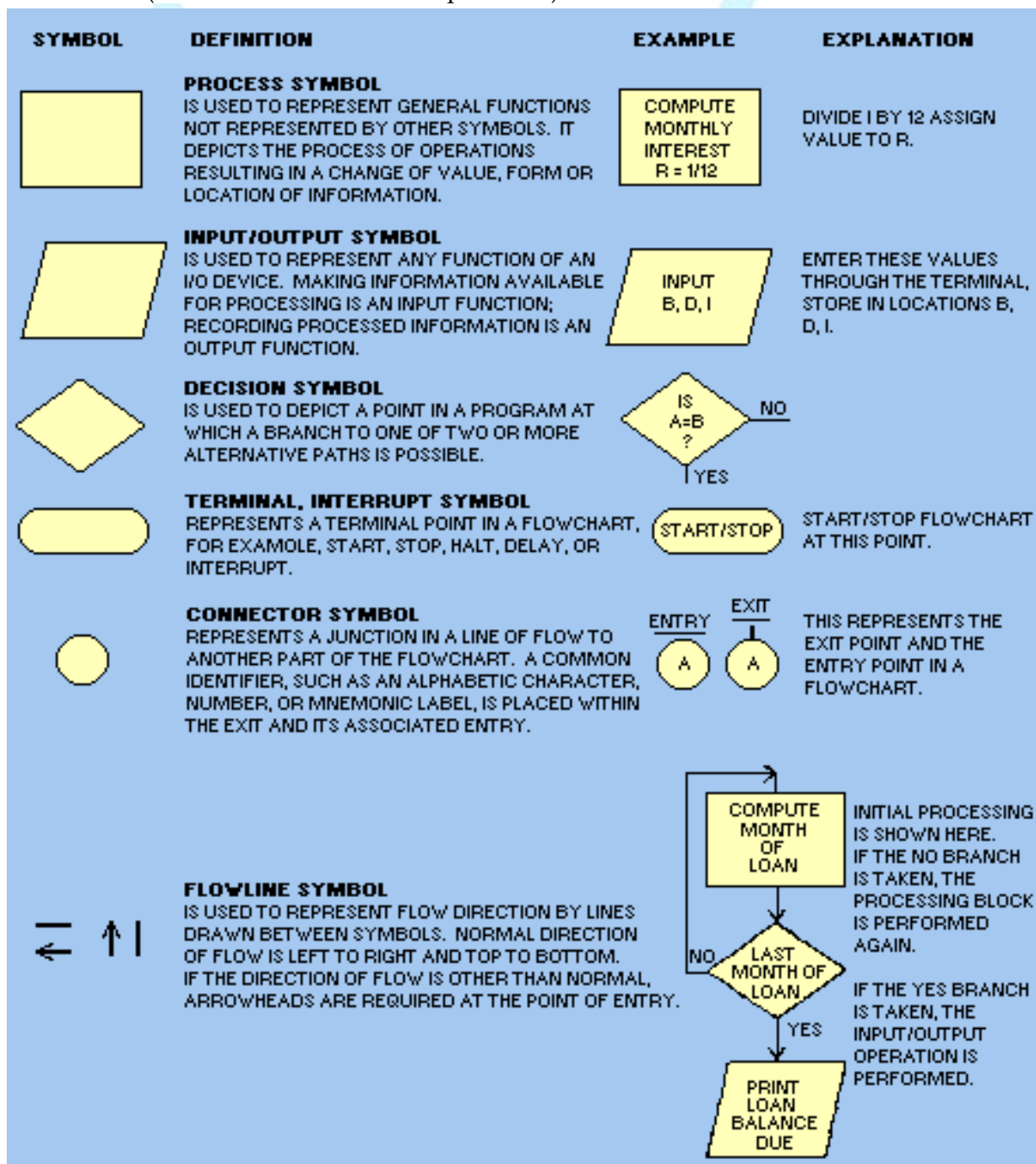
```
Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Input variables a, b and c.
Step 4: If a>b
    If a>c
        Display a is the largest number.
```

```

Else
    Display c is the largest number.
Else
    If b>c
        Display b is the largest number.
    Else
        Display c is the greatest number.
Step 5: Stop
    
```

### Flowchart

A graphical representation of a computer program in relation to its sequence of functions (as distinct from the data it processes).



### Advantages of flowchart:

1. The Flowchart is an excellent way of communicating the logic of a program.
2. It is easy and efficient to analyze problem-using flowchart.
3. During program development cycle, the flowchart plays the role of a guide or a blueprint. Which makes program development process easier.
4. After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
5. It helps the programmer to write the program code.
6. It is easy to convert the flowchart into any programming language code as it does not use any specific programming language concept.

### Disadvantage of flowchart

1. The flowchart can be complex when the logic of a program is quite complicated.
2. Drawing flowchart is a time-consuming task.
3. Difficult to alter the flowchart. Sometimes, the designer needs to redraw the complete flowchart to change the logic of the flowchart or to alter the flowchart.
4. Since it uses special sets of symbols for every action, it is quite a tedious task to develop a flowchart, as it requires special tools to draw the necessary symbols.
5. In the case of a complex flowchart, other programmers might have a difficult time understanding the logic and process of the flowchart.
6. It is just a visualization of a program; it cannot function like an actual program.

### Instruction

A basic command/order. An instruction is a segment of code that contains steps that need to be executed by the computer processor. For a computer to do something, instructions have to be given to the computer processor so it knows how to do what it is being asked to do.

**For example,** if you were to ask the computer to draw a square, it would need a set of instructions on how to draw the square, so it can complete the task.

### Computer Program

A computer program is a collection of instructions that performs a specific task when executed by a computer. A computer requires programs to function and typically executes the program's instructions in a central processing unit. A computer program is usually written by a computer programmer in a programming language.

### **Programming**

It is a process of writing set of rules/instructions that provides a way of telling a computer what operations to perform.

### **Programmer**

A computer programmer, or coder, is someone who writes computer software. The term computer programmer can refer to a specialist in one area of computer programming or to a generalist who writes code for many kinds of software.

### **Role of a Programmer**

The programmer's job is to convert problem solutions into instructions for the computer. That is, the programmer prepares the instructions of a computer program and runs those instructions on the computer, tests the program to see if it is working properly, and makes corrections to the program. The programmer also writes a report on the program. These activities are all done for the purpose of helping a user fill a need, such as paying employees, billing customers, or admitting students to college.

### **Programming Language**

All the human beings in this world communicate with each other by a language. Similarly, in order to communicate with computer user also needs a language that should be understandable by computers.

PL is an interface between human & computer they allow us to communicate with computer in order to perform task. This are used to develop system and applications software's.

A set of rules that provides a way of telling a computer what operations to perform is called a programming language.

### **Types of Computer language**

1. Low level language/Machine Level/1<sup>st</sup> GL
2. Assembly Level Language/Middle Level Language/2<sup>nd</sup> GL.
3. High Level Language/3<sup>rd</sup> GL.
4. 4<sup>th</sup> Generation Language.

#### **Low Level Language:**

Low level languages are the machine codes in which the instructions are given in machine language in the form of 0 and 1 to a Computer system. It is mainly designed to operate and handle all the hardware and instructions set architecture of a Computer. The main function of the Low level language is to operate, manage and manipulate the hardware and system components.

### **Middle Level or Assembly Language**

The second generation programming language that has almost similar structure and set of commands as Machine language. Instead of using numbers like in Machine languages here we use words or names in English forms and also symbols. The programs that have been written using words, names and symbols in assembly language are converted to machine language using an Assembler. Because a Computer only understands machine code languages that's why we need an Assembler that can convert the Assembly level language to Machine language so the Computer gets the instruction and responds quickly.

The main disadvantage of this language is that it is written only for a single type of CPU and does not run on any other CPU. But its speed makes it the most used low level language till today which is used by many programmers.

### **High Level Language:**

The high level languages are the most used and also more considered programming languages that helps a programmer to read, write and maintain. It is also the third generation language that is used and also running till now by many programmers. They are less independent to a particular type of Computer and also require a translator that can convert the high level language to machine language. The translator may be an interpreter and Compiler that helps to convert into binary code for a Computer to understand. There is various high level programming languages like **C**, **FORTTRAN** or Pascal that are less independent and also enables the programmer to write a program.

### **4<sup>th</sup> Generation Language**

4GLs are closer to human language than other high-level languages and are accessible to people without formal training as programmers. They allow multiple common operations to be performed with a single programmer-entered command. They are intended to be easier for users than machine languages (first-generation), assembly languages (second-generation), and older high-level languages (third-generation). 4GLs are designed to reduce the overall time, effort and cost of software development.

### **Overview of C language**

C is a **structured, general purpose, high level programming language** developed by American programmer **Dennis M. Ritchie** in the year **1972** at **AT & T's** (American Telephone & Telegraph) **Bell Laboratories**. C is a **powerful, flexible, portable** programming language. It is one of the most popular computer languages today because of its structure, high-level abstraction, machine independent feature. C language was developed with **UNIX operating system**, so it is strongly associated



with UNIX, which is one of the most popular network operating system in use today and heart of internet data superhighway.

C programming is considered as the base for other programming languages, that is why it is known as **mother language**.

**It can be defined by following ways:**

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

### History of C Programming Language

C language has evolved from three different structured language **ALGOL** (Algorithmic Language), **BCPL** (Basic Combined Programming Language) and **B** Language. Both **BCPL & B** were “**typeless**” languages. Basically they are known as **low level languages**. It uses many concepts from these languages and introduced many new concepts such as **data types, struct, and pointer**. In 1988, the language was formalised by American National Standard Institute (ANSI). In 1990, a version of C language was approved by the International Standard Organisation (ISO) and that version of C is also referred to as **C89, C99 & C11 (C1X)** is the latest version of C language.

1960	• Algol	• International Group
1967	• BCPL	• Martin Richards
1970	• B	• Ken Thomson
1972	• Traditional C	• Dennis Ritchie
1978	• K & RC	• kernighan & Ritchie
1989	• ANSI C	• ANSI Commitee
1990	• ANSI / ISO C	• ISO Commitee
1999	• C99	• Standerd Commitee

### Why Name "C" was given to Language?

- ✓ Many of C's principles and ideas were derived from the earlier language B. (Ken Thompson was the developer of B Language.)
- ✓ BCPL and CPL are the earlier ancestors of B Language
- ✓ CPL is common Programming Language. In 1967, BCPL Language ( Basic CPL ) was created as a scaled down version of CPL
- ✓ As many of the **features were derived from "B" Language that's why it was named as "C"**.
- ✓ After 7-8 years, C++ came into existence, which was first example of object-oriented programming.

### Importance/Characteristics or features of C Language

#### 1. **Modular Programming**

- a. Modular programming is a software design technique that increases the extent to which software is composed of separate parts, called **Modules**.
- b. C Program consist of different modules that are integrated together to form complete program.

#### 2. **Case sensitivity:**

- a. C is a case sensitive language. It means that it can differentiate uppercase and lowercase words.

#### 3. **Machine independent:**

- a. It is a machine independent language. It means that the program written in C language can be executed on different types of computers.

#### 4. **Hardware control:**

- a. C programming language provides close control on hardware. It can be used to write efficient programs to control hardware components of computer system.

#### 5. **Small language:**

- a. C is a small language. It has a small number of keywords and programming controls. But still it is very powerful for developing different types of programs.

#### 6. **Portability:**

- a. C Programs are portable i.e. they can run on any Compiler with little or No Modification.
- b. Compiler and Pre-processor make it possible for C Programs to run it on Different Platforms.

#### 7. **More Efficient:**

- a. C Programming language is more efficient to other languages.

**8. Powerful:**

- a. Provides wide verity of “Data Types”
- b. Provides wide verity of “Functions”
- c. Provides useful Control and Loop Control Statements.

**9. Fast code generation:**

- a. The compilers of C language generate very fast code. The code executes very efficiently. So the programs take less time to execute.

**10. Extensibility:**

- a. C language has the ability to extend itself, it is the collection of functions which are supported by the C library this makes us easier to add our own functions to C library, Because of the availability of large number of functions, the programming task becomes simple.

**11. Easy to Learn:**

- a. C programming language is easy to learn (Syntax is near to English Language), C language syntax is very easy to understand, and it uses keywords that we are using in our day-to-day life to convey meaning or to get some decisions.

**12. POP: Procedure Oriented Paradigm**

- a. C programming language is procedure oriented language, so, the user creates procedures or functions to execute their task, the procedure-oriented language is very much easy to learn because it follows algorithm to execute your statements.

**13. Bit Manipulation:**

- a. C Programs can be manipulated using bits. We can perform different operations at bit level.
- b. It provides wide verity of bit manipulation Operators. We have bitwise operators to manage Data at bit level.

**14. Efficient Use of Pointers:**

- a. Pointers have direct access to memory.
- b. C supports efficient use of pointer.

नहि ज्ञानेन सद्रशं  
SHREE MEDHA DEGRE COLLEGE



### **1. A sample C Program:**

Below C program is a very simple and basic program in C programming language. This C program displays "Hello World!" in the output window. And, all syntax and commands in C programming are case sensitive. Also, each statement should be ended with semicolon (;) which is a statement terminator.

```
1 #include <stdio.h>
2 int main()
3 {
4     /* Our first simple C basic program */
5     printf("Hello World! ");
6     getch();
7     return 0;
8 }
```

#### **Output:**

Hello World!

### **2. Steps to write C programs and get the output:**

Below are the steps to be followed for any C program to create and get the output. This is common to all C program and there is no exception whether its a very small C program or very large C program.

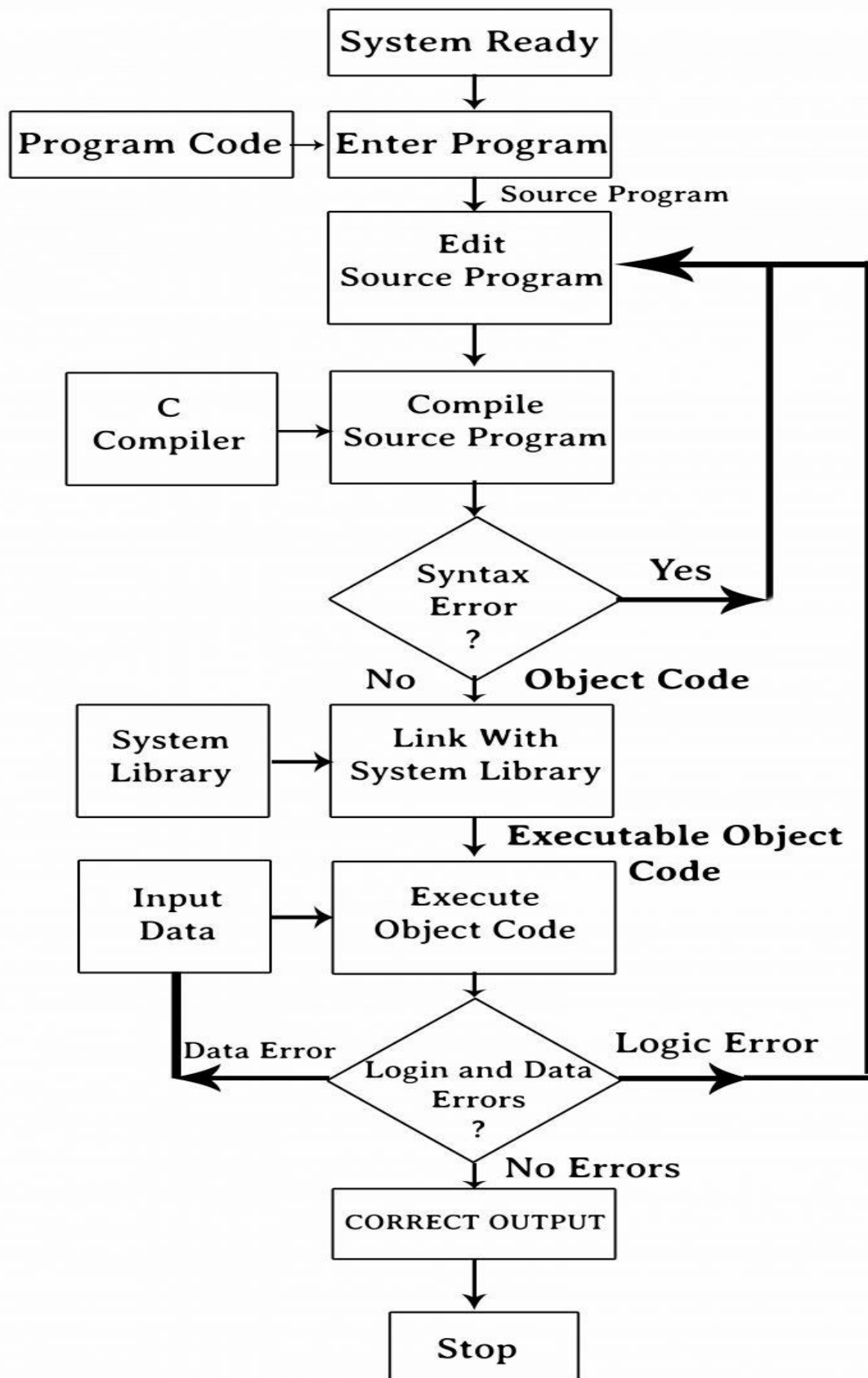
1. Create
2. Compile
3. Execute or Run
4. Get the Output

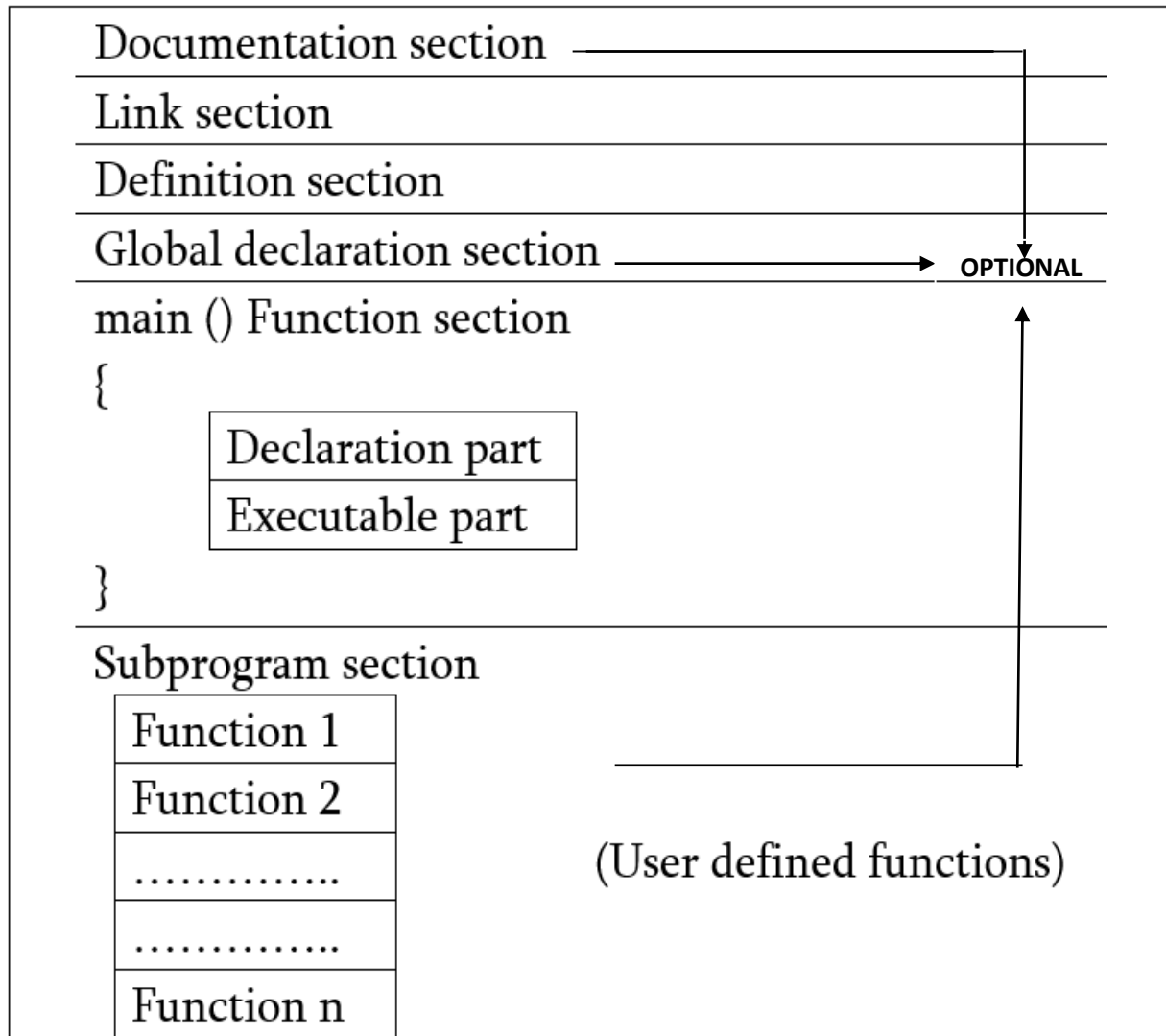
### **3. Creation, Compilation and Execution of a C program:**

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps –

- Open a TC editor and add the above-mentioned code.
- Save the file as *hello.c*
- Open a command prompt and go to the directory where you have saved the file.
- Press **Alt + F9** enter to compile your code.
- If there are no errors in your code, the command prompt will take you to the next line and would generate *hello.obj* executable file.
- Now, press **Ctrl+F9** to execute your program.
- You will see the output "Hello World" printed on the screen.

## Process Of Compiling and Running Of A C Program



Structure of C Program**Documentation Section**

The documentation section contains a set of comment lines which contain the name of the program other necessary details. Comment lines are used for readability and understand ability. Comments are ignored by compiler and are used to provide documentation to people who reads that code. Documentation section helps anyone to get an overview of the program. Comment lines are of two types:

**1. Single Line Comment**

Comments are non-executable code used to provide documentation to programmer. Single Line Comment is used to comment out just Single Line in the Code. It is used to provide One Liner Description of line.

**Some Important Points:**

1. Single Line Comment Can be **Placed Anywhere**
2. Single Line Comment **Starts with '//'**
3. Any Symbols written after **'//'** are **ignored by Compiler**.
4. Comment cannot hide statements written before **'//'** and On the Successive new line

//This is single line comment

**2. Multi Line Comment**

Multiple Line Comment is used to comment in more than one Line in the Code. It is used to provide Multi liner Description of line.

### ***Multi Line Comment***

1. Multi line comment can be placed anywhere.
2. Multi line comment starts with /\*.
3. Multi line comment ends with \*/.
4. Any symbols written between '/\*' and '\*/' are ignored by Compiler.
5. It can be split over multiple lines

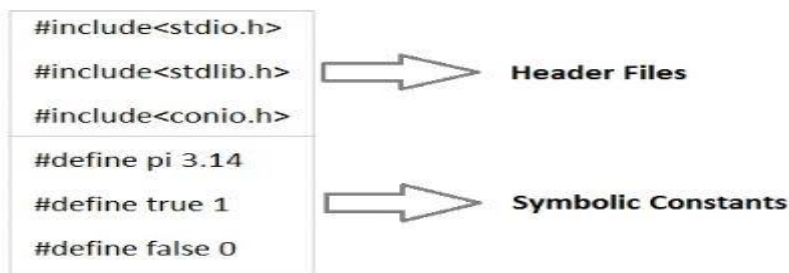
```
/*This is  
multi line  
comment */
```

### ***Link Section or Pre-processor Section & Definition Section***

The link section provides instructions to the compiler to link functions from the system library such as using the #include directive.

- This section is comprised of both link section and definition section
- This section mainly contains Header files and Pre-processors. This link passes instructions to the compiler to link files or functions or pre-processors from the system library.
- The pre-processor command starts with a symbol “#”. C Pre-processors are not a part of a compiler, but they are separate steps in the compilation process.
- C pre-processor may include symbolic constants also. Definition Section may include all the symbolic constants defined with the #define directive.
- For Executing a C program, we must include Header files in this Link section. Each and Every Header file is a file with Extension .h.

Example:



### ***Global Declaration Section***

- Many of the variables may be used in more than one function. Such variables are known as Global variables. Global variables are always declared above the main ( ). They must be declared in this Global Declaration section.
- Global variables may be Constant, Static, Extern..., etc.
- We can include User-defined functions and User-defined data types here.

### ***Syntax:***

<datatype> variable name;

### ***Example:***

```
const int a=10;  
int b=10;
```

In the above example, a and b are two Global variables which may be used anywhere in the C program. 'a' value is constant throughout the program while 'b' value changes.

***main() Function Section:***

- Each and Every C program must have only one main function. Program Execution always starts from the main() function only.
- The main() should be written in small letters only since it is case sensitive. The main() function should be opened and closed with the left and right flower braces( { and } ).
- The main function shouldn't be terminated with colon(:) or semi colon(;) .
- main() function may contains Local variables, Global variables, User-defined functions....etc.
- Execution Process starts from main() function, and it may call other User-defined functions or System defined functions.
- The return type of the main function is of any type in C. If it is void we don't include the return statement.

***Subprogram section:***

- This section contains all the User-defined functions that are called and declared in the other functions like main().
- User-defined functions must be placed immediately after the main() function even they may appear in order.
- This section may or may not be included in the C program. It is optional.
- These functions perform user tasks, and this also contains a set of program statements.

Sample C Program	Explanation
#include <stdio.h>	This is a pre-processor command that includes standard input output header file(stdio.h) from the C library before compiling a C program
void main()	This is the main function from where execution of any C program begins. void means returning nothing.
{	This indicates the beginning of the main function.
/*_some_comments_*/	whatever is given inside the command "/* */" in any C program, won't be considered for compilation and execution.
printf("Hello_World! ");	printf command prints the output onto the screen.
getch();	This command waits for any character input from keyboard.
}	This indicates the end of the main function.



***C Programming Introduction: Tips***

1. Every C Program Should have exactly **one main function**
2. C Program **Execution always starts from main.**
3. Execution of C Program **begins at Opening brace of function** and ends at **closing brace of the function.**
4. Generally, all statements in c are written in **Lowercase Letters.**
5. Uppercase Letters are used for **Symbolic** names, output strings and messages
6. Every C statement **must end with semicolon**
7. All variables must be declared with respective **data types** before using.
8. C is **free form-Language**
9. Comments can be **inserted anywhere in C Program**, but nested comments are not supported by C.
10. **Braces** are Generally Used for the **Grouping of statements**

***Applications of C Programming***

C Programming is best known programming language. C Programming is near to machine as well as human so it is called as **Middle level Programming Language**. C Programming can be used to do verity of tasks such as networking related, OS related.

***Application of C Programming are listed below -***

1. C language is used for creating **computer applications**
2. Used in writing **Embedded software's**
3. Firmware for various electronics, industrial and communications products which use micro-controllers.
4. It is also used in developing **verification software, test code, simulators** etc. for various applications and hardware products.
5. **For Creating Compiles** of different Languages which can take input from other language and convert it into lower level machine dependent language.
6. C is used to implement different **Operating System Operations.**
7. **UNIX kernel** is completely developed in C Language.

***List of Applications of C Programming***

List of Application		
Operating Systems	Network Drivers	Print Spoolers
<u>Language Compilers</u>	Assemblers	Text Editors
Modern Programs	Data Bases	Language Interpreters
Simulators	Utilities	Embedded System

### Character set of C

**character:** - It denotes any alphabet, digit or special symbol used to represent information.

**Use:** - These characters can be combined to form variables. C uses constants, variables, operators, keywords and expressions as building blocks to form a basic C program.

**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.

**Character Set Consists of -**

Types	Character Set
Lowercase Letters	a-z
Uppercase Letters	A to Z
Digits	0-9
Special Characters	!@#\$%^&*
White Spaces	Tab Or New line Or Space

### C Tokens

- In a C source file, each word, and punctuations mark are called as Tokens.
- As we know, some the smallest individual units in a program are known as Tokens.
- Tokens are also called as Lexical Units.
- These are the building blocks in C language which are used to write a program.

**C Tokens can be classified as follows:**

1. Keywords (eg: int, while),
2. Identifiers (eg: main, total),
3. Constants (eg: 10, 20),
4. Strings (eg: "total", "hello"),
5. Special symbols (eg: {}, {}),
6. Operators (eg: +, /, -, \*)

**Keywords or Reserve words**

- Every C word is classified as either a Keyword or an Identifier.
- All Keywords in C have fixed to mean, and these meanings can't be changed anywhere in the C program.
- Keywords play a vital role in Programming, and these are the essential building blocks to write a C program.
- There are 32 keywords in C language. Keywords are pre-defined words in the compiler.
- The keywords cannot be used as variable names because they assign a new meaning to the keyword which is not allowed.
- All the keywords must be written in lowercase only.
- Some compilers may use additional keywords that must be identified from the C manual.

**List of Keywords:**

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

**Identifier or Variables**

Identifiers are the names that are given to various program elements such as variables, symbolic constants and functions. Variable or function identifier that is called a symbolic constant name.

**Rules for using identifiers**

1. First character should be an alphabet or underscore ( \_ ).
2. Succeeding characters might be digits or letter.
3. Punctuation and special characters are not allowed except underscore.
4. A keyword cannot be used as a variable name or an identifier name. (such as int , float, if ,break, for etc)
5. Identifiers are also case sensitive in C. For example, name and Name are two different identifiers in C.
6. Successive underscores are not allowed( \_\_ ).

**Valid Names**

num  
Num  
Num1  
\_NUM  
NUM\_temp2  
Int

### Invalid Identifiers

1num  
1\_num  
365\_days

**Reason: First Character must be underscore or Alphabet**

number 1  
num 1  
addition of program

**Reason : blanks are not allowed**

int  
char  
continue

**Reason: Reserve words are not allowed**

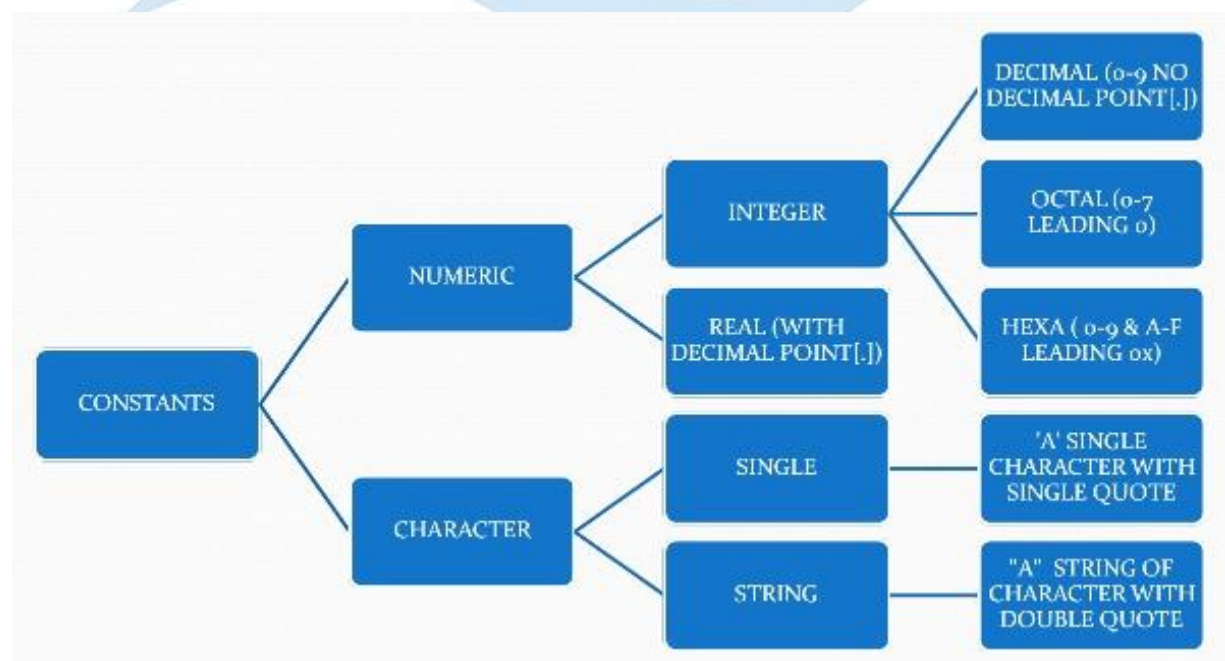
### Constants or Literals

Constants refer to fixed values that don't change during its execution. These fixed values are also called **literals**. **DOB** is considered as constant.

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

### Types of Constants



## **Integer Constants**

An integer constant is a sequence of digits from 0 to 9 without decimal points or fractional part or any other symbols. The number may be positive or negative. These constants are combination of digits from 0 to 9.

*There are 3 types of integers namely decimal integer, octal integers and hexadecimal integer.*

**Decimal Integers** consists of a set of digits 0 to 9 preceded by an optional + or - sign. Spaces, commas and non-digit characters are not permitted between digits. Example for valid decimal integer constants are

**Decimal constants: int a=0, b=-9, c= 22 etc**

**Octal Integers** constant consists of any combination of digits from 0 through 7 with a O at the beginning. Some examples of octal integers are

**Octal constants: int a=021, b=077, c=033 etc**

**Hexadecimal integer** constant is preceded by OX or Ox, they may contain alphabets from A to F or a to f. The alphabets A to F refers to 10 to 15 in decimal digits. Example of valid hexadecimal integers are

**Hexadecimal constants: int a=0x7f, b=0x2a, c=0x521 etc**

## **Real Constants**

Real Constants consists of a fractional part in their representation. Integer constants are inadequate to represent quantities that vary continuously. These values are represented by numbers containing fractional parts like 26.082. Example of real constants are

**float x=6.3;** // here 6.3 is a double constant.

**float y=6.3f;** // here 6.3f is a float constant.

**float z=6.3e+2;** // here 6.3 e + 2 is a exponential constant.

**float s = 6.3L ;** // here 6.3L is a long double constant

Real Numbers can also be represented by exponential notation. The general form for exponential notation is mantissa exponent. The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign.

## **Single Character Constants**

A Single Character constant represent a single character, which is enclosed in a pair of quotation symbols.

*Example for character constants are*

**char y ='u';** // y will hold the value 'u'

**char k ='34' ;** // k will hold the value '3, and '4' will be omitted

**char p ='ok' ;** // p will hold the value 'O' and k will be omitted

**char e =' ';** // e will hold the value ' ', a blank space

**chars ='\\45';** // swill hold the value ' ', a blank space

All character constants have an equivalent integer value, which are called **ASCII** (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE) Values.



**String Constants**

A string constant is a set of characters enclosed in double quotation marks. The characters in a string constant sequence may be a alphabet, number, special character and blank space. Example of string constants are

"VISHAL" "1234" "God Bless" "!!.....?"

**Backslash Character Constants [Escape Sequences]**

Backslash character constants are special characters used in output functions. Although they contain two characters they represent only one character. Given below is the table of escape sequence and their meanings.

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

## Datatypes

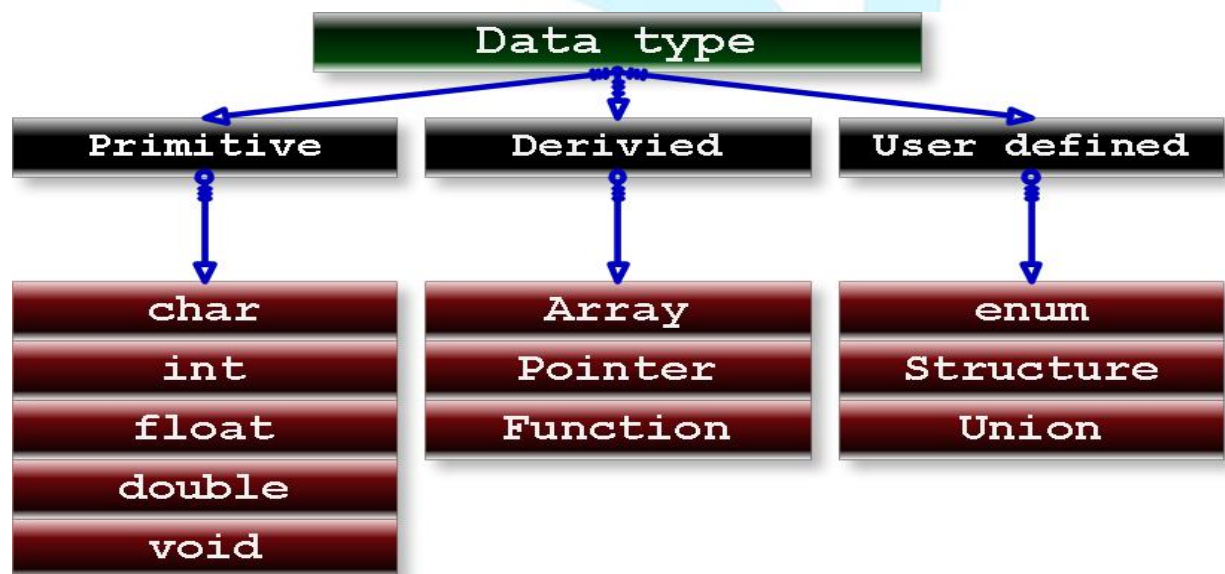
### *What is Data Type in C Programming?*

1. A Data Type is a **Type of Data**.
2. Data Type is a Data Storage Format that can contain a **Specific Type or Range of Values**.
3. When computer programs store data in variables, each variable must be assigned a **specific data type**.
4. Datatype is a set of value with predefined characteristics. Datatypes are used to declare variable, constants, arrays, pointers and functions.

### *Explanation:*

1. Whenever we declare variable in Computer's memory, Computer must know the **type of the data to be stored inside the memory**.
2. If we need to store the single character, then the size of memory occupied will be different than storing the single integer number.
3. The memory in our computers is organized in bytes. A **byte is the minimum amount of memory** that we can manage in C.
4. A byte can store a relatively small amount of data one single character or a small integer (generally an integer between 0 and 255).

### Different Data Types



### *Basic Data Types or Primary or Fundamental or Pre Defined*

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

#### *int - Integer data types*

Integers are whole numbers that can have both positive and negative values but no decimal values. The size of int is **2 bytes**. The range of integer is **-32,768 to 32,767**.

**Example:** 0, -5, 10

In C programming, keyword *int* is used for declaring integer variable. For example:  
*int id;*

Here, *id* is a variable of type integer.

You can declare multiple variable at once in C programming. For example:

*int id, age;*

### *float - Floating types*

Floating type variables can hold real numbers such as: 2.34, -9.382, 5.0 etc. You can declare a floating point variable in C by using *float* keyword. The size of *float* (single precision float data type) is 4 bytes. Floating point variables has a precision of 6 digits.

**For example:**

***float accountBalance;***

***float bookPrice;***

Here, both *accountBalance* and *bookPrice* are floating type variables.

In C, floating values can be represented in exponential form as well. For example:

*float normalizationFactor = 22.442e2;*

### *char - Character types*

Keyword *char* is used for declaring character type variables. For example:

*char test = 'h';*

Here, *test* is a character variable. The value of *test* is 'h'.

The size of character variable is 1 byte.

### *double - double precision types*

Double type variables can hold real numbers such as: 2.335874, -9.382124, 5.00054 etc. You can declare a floating point variable in C by using *double* keyword. The size of *double* (double precision float data type) is 8 bytes. Double variables has a precision of 14 digits.

***double accountBalance;***

***double bookPrice;***

### *Data Type Qualifiers*

Apart from the primitive data types mentioned above, there are certain data type qualifiers that can be applied to them in order to alter their range and storage space and thus, fit in various situations as per the requirement.

The data type qualifiers available in c are:

- short
- long
- signed
- unsigned

**Size qualifiers**

Size qualifiers alter the size of a basic type. There are two size qualifiers, *long* and *short*. For example:

*long double i;*

The size of *double* is 8 bytes. However, when *long* keyword is used, that variable becomes 10 bytes.

Learn more about [long keyword in C programming](#).

There is another keyword *short* which can be used if you previously know the value of a variable will always be a small number.

**Sign qualifiers**

Integers and floating point variables can hold both negative and positive values. However, if a variable need to hold positive value only, *unsigned* data types are used. For example:

*// unsigned variables cannot hold negative value*

*unsigned int positiveInteger;*

There is another qualifier *signed* which can hold both negative and positive only. However, it is not necessary to define variable *signed* since a variable is signed by default.

An integer variable of 4 bytes can hold data from  $-2^{31}$  to  $2^{31}-1$ . However, if the variable is defined as unsigned, it can hold data from 0 to  $2^{32}-1$ .

It is important to note that, sign qualifiers can be applied to *int* and *char* types only.

Let's see the basic data types. Its size is given according to 32-bit architecture.

	Integer	Floating Point	Double	Character
What is it?	Numbers without decimal value	Numbers with decimal value	Numbers with decimal value	Any symbol enclosed in single quotation
Keyword	int	float	double	char
Memory Size	2 or 4 Bytes	4 Bytes	8 or 10 Bytes	1 Byte
Range	-32768 to +32767 (or) 0 to 65535 (Incase of 2 bytes only)	1.2E - 38 to 3.4E + 38	2.3E-308 to 1.7E+308	-128 to + 127 (or) 0 to 255
Type Specifier	%d or %i or %u	%f	%ld	%c or %s
Type Modifier	short, long signed, unsigned	No modifiers	long	signed, unsigned
Type Qualifier	const, volatile	const, volatile	const, volatil	const, volatile

### *Derived Datatypes*

Derived datatypes are user-defined data types. The derived datatypes are also called as user defined datatypes or secondary datatypes. In c programming language, the derived datatypes are created using the following concepts...

- Arrays
- Structures
- Unions
- Enumeration

### **void Datatype**

The void datatype means nothing or no value. Generally, void is used to specify a function which does not return any value. We also use the void datatype to specify empty parameters of a function.

### **Enumerated Datatype**

An enumerated datatype is a user-defined data type that consists of integer constants and each integer constant is given a name. The keyword "**enum**" is used to define enumerated datatype.

```
enum flag { const1, const2, ..., constN };
```

Here, name of the enumeration is flag.

And, const1, const2,..., constN are values of type flag.

By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary).

```
// changing default values of enum
```

```
enum suit {  
    club = 0,  
    diamonds = 10,  
    hearts = 20,  
    spades = 3,  
};
```

### *Enumerated Type Declaration*

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };  
enum boolean check;
```

Here, a variable check of type enum boolean is created.

Here is another way to declare same check variable using different syntax.

```
enum boolean  
{  
    false, true  
} check;
```



### Example: Enumeration Type

```
#include <stdio.h>
enum week { sunday, monday, tuesday, wednesday, thursday, friday, saturday };
void main()
{
    enum week today;
    today = wednesday;
    printf("Day %d",today+1);
}
```

#### Output

Day 4

### Why enums are used in C programming?

Enum variable takes only one value out of many possible values. Example to demonstrate it,

```
#include <stdio.h>
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3
} card;

void main()
{
    card = club;
    printf("Size of enum variable = %d bytes", sizeof(card));
}
```

#### Output

Size of enum variable = 4 bytes

It's because the size of an integer is 4 bytes.

This makes enum a good choice to work with flags.

You can accomplish the same task using structures. However, working with enums gives you efficiency along with flexibility.

### How to use enums for flags?

Let us take an example,

```
enum designFlags {
    ITALICS = 1,
    BOLD = 2,
    UNDERLINE = 4
} button;
```

Suppose you are designing a button for Windows application. You can set flags ITALICS, BOLD and UNDERLINE to work with text.

There is a reason why all the integral constants are power of 2 in above pseudocode.

```
// In binary
ITALICS = 00000001
BOLD = 00000010
UNDERLINE = 00000100
```

Since, the integral constants are power of 2, you can combine two or more flags at once without overlapping using bitwise OR | operator. This allows you to choose two or more flags at once. For example,

```
#include <stdio.h>
enum designFlags {
    BOLD = 1,
    ITALICS = 2,
    UNDERLINE = 4
};

int main() {
    int myDesign = BOLD | UNDERLINE;
    // 00000001
    // | 00000100
    // -----
    // 00000101
    printf("%d", myDesign);
    return 0;
}
```

Output  
5

When the output is 5, you always know that bold and underline is used. Also, you can add flag to your requirements.

```
if (myDesign & ITALICS) {
    // code for italics
}
```

Here, we have added italics to our design. Note, only code for italics is written inside if statement.

You can accomplish almost anything in C programming without using enumerations. However, they can be pretty handy in certain situations. That's what differentiates good programmers from great programmers.

## Variables

Variable is a value which changes during the execution of program. Variables in c programming language are the named memory locations where user can store different values of same datatype during the program execution. That means, variable is a name given to a memory location in which we can store different values of same datatype. In other words, a variable can be defined as a storage container to hold values of same datatype during the program execution. The formal definition of datatype is as follows...

Variable is a name given to a memory location where we can store different values of same datatype during the program execution.

Every variable in c programming language must be declared in the declaration section before it is used. Every variable must have a datatype that determines the range and type of values to be stored and size of the memory to be allocated.

A variable name may contain letters, digits and underscore symbol. The following are the rules to specify a variable name...

1. Variable name should not start with digit.
2. Keywords should not be used as variable names.
3. Variable name should not contain any special symbols except underscore (\_).
4. Variable name can be of any length but compiler considers only the first 31 characters of the variable name.

## Declaration of Variable

Declaration of a variable tells to the compiler to allocate required amount of memory with specified variable name and allows only specified datatype values into that memory location. In C programming language, the declaration can be performed either before the function as global variables or inside any block or function. But it must be at the beginning of block or function.

### **Declaration Syntax:**

*datatype variableName;*

### **Example**

```
int i, j, k;  
char c, ch;  
float f, salary;  
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

**Initializing or Assigning values to variables** Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows –

***datatype variable\_name = value;***

Some examples are –

```
extern int d = 3, f = 5; // declaration of d and f.  
int d = 3, f = 5;      // definition and initializing d and f.  
byte z = 22;          // definition and initializes z.  
char x = 'x';         // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

### **Symbolic Constant**

A symbolic constant is a symbol, often similar to a variable name, which represents one and only one value. You can think of them as variables that never change (not while the program is running anyway). Often times in a program you might want to use a single value for something over and over. Well, that is what symbolic constants are for. Before I go on, let us see how to make a symbolic constant.

A **C program** consists of the following symbolic constant definitions.

```
#define PI 3.141593  
#define TRUE 1  
#define FALSE 0
```

*#define PI 3.141593* defines a symbolic constant PI whose value is 3.141593. When the program is pre-processed, all occurrences of the symbolic constant PI are replaced with the replacement text 3.141593.

Note that the **pre-processor statements** begin with a #symbol, and are not end with a semicolon. By convention, **pre-processor** constants are written in UPPERCASE.

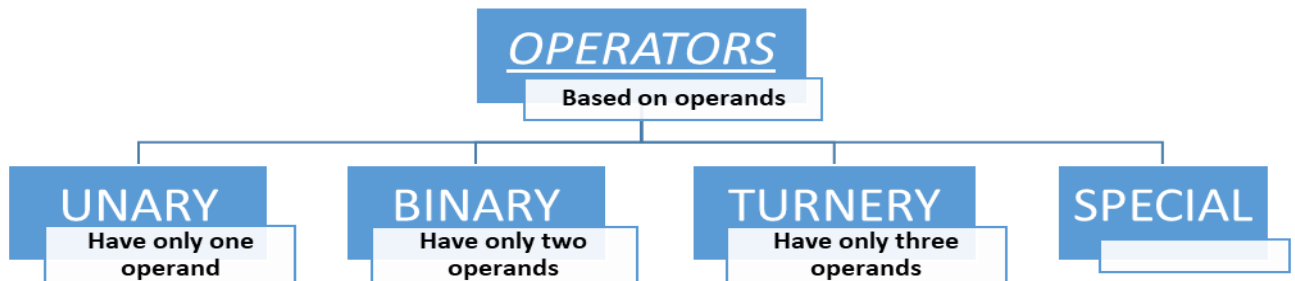
नाह ज्ञानन सद्रश  
SHREE MEDHA DEGRE COLLEGE

## C Programming Operators

An operator is a **symbol** that tells the compiler to perform **specific mathematical or logical** operations.

C programming language has wide range of operators to perform various operations. For better understanding of operators, these operators can be classified as:

### Types of Operators



### Unary Operator

A unary operator, in C, is an operator that takes a single operand in an expression or a statement.

### Types of Unary Operator

1. Unary Minus (-)
2. Increment (++)
3. Decrement (--)

### Unary Minus (-)

The - (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if quality has the value 100, -quality has the value -100.

### Increment and decrement operators

**Increment Operators** are used to increase the value of the variable by one and **Decrement Operators** are used to decrease the value of the variable by one in C programs.

Both increment and decrement operator are used on a **single operand or variable**, so it is called as a **unary operator**. Unary operators are having higher priority than the other operators it means unary operators are executed before other operators.

### Increment operator (++)

1. Increment operator is used to increment the current value of variable by adding integer 1.
2. Increment operator can be applied to only variables.
3. Increment operator is denoted by ++.

### Type of Increment Operator

- pre-increment
- post-increment



***pre-increment (++ variable)***

In pre-increment first increment the value of variable and then used inside the expression (initialize into another variable).

**Syntax**

<code>++ variable;</code>
---------------------------

***post-increment (variable ++)***

In post-increment first value of variable is used in the expression (initialize into another variable) and then increment the value of variable.

**Syntax**

<code>variable ++;</code>
---------------------------

***Decrement operator (--)***

1. Decrement operator is used to decrease the current value of variable by subtracting integer 1.
2. Like Increment operator, decrement operator can be applied to only variables.
3. Decrement operator is denoted by `--`.

***Type of Decrement Operator***

1. pre-decrement
2. post-decrement

***Pre-decrement (-- variable)***

In pre-decrement first decrement the value of variable and then used inside the expression (initialize into another variable).

**Syntax**

<code>-- variable;</code>
---------------------------

***post-decrement (variable --)***

In Post-decrement first value of variable is used in the expression (initialize into another variable) and then decrement the value of variable.

**Syntax**

<code>variable --;</code>
---------------------------

**Difference between ++ and -- operator as postfix and prefix**

When `++` is used as prefix (like: `++var`), `++var` will increment the value of `var` and then return it but, if `++` is used as postfix (like: `var++`), operator will return the value of operand first and then only increment it. This can be demonstrated by an example:

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

```
void main()
```

```
{
```

```
    int c=2,d=2;
```

```
    printf("%d\n",c++); //this statement displays 2 then, only c incremented by 1 to 3.
```

```
    printf("%d",++c); //this statement increments 1 to c then, only c is displayed.
```

```
    getch();
```

```
}
```

**Output** 2 4

## Binary Operators

### Arithmetic Operator

Operator	Description
*	multiplication
/	division
%	modulo
+	addition
-	subtraction

### Logical Operator

Operator	Description
!	NOT
&&	AND
	OR

### Compound Assignment

Operator	Name
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Float division assignment

**=**                      **Assignment Operator**

### Relational Operator

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

### Bitwise Operators

Operator	Description
~	One's complement
<<	Left shift
>>	Right shift
&	Bitwise AND
^	Bitwise XOR

## Arithmetic Operators

Arithmetic operators, in C, are operators used to perform arithmetic operations that include multiplication, division, addition and subtraction. All these operators are binary operators which means they operate on two operands.

Following table shows all the arithmetic operators supported by C language. Assume variable **A=10** and variable **B=20**, then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

**Example of working of arithmetic operators**

```
/* Program to demonstrate the working of arithmetic operators in C. */
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,add,sub,mul,mod;
    float div;
    clrscr();
    printf("Enter the values of a & b\n");
    scanf("%d%d",&a,&b);
    add=a+b;
    sub=a-b;
    mul=a*b;
    div=(float)a/(float)b;
    mod=a%b;
    printf("Addition of two numbers=%d\n",add);
    printf("Subtraction of two numbers=%d\n",sub);
    printf("Multiplication of two numbers=%d\n",mul);
    printf("Division of two numbers=%f\n",div);
    printf("Modulus of two numbers=%d\n",mod);
    getch();
}
```

---

**OUTPUT**

---

```
Enter the values of a & b
15
10
Addition of two numbers=25
Subtraction of two numbers=5
Multiplication of two numbers=150
Division of two numbers=1.5
Modulus of two numbers= 0
```

**Explanation**

Here, the operators +, - and \* performed normally as you expected. In normal calculation, 15/10 equals to 1.5. And, finally a%b is 0, i.e., when a=15 is divided by b=10, remainder is 0.

Suppose a=5.0, b=2.0, c=5 and d=2

**In C programming,**

```
a/b=2.5
a/d=2.5
c/b=2.5
c/d=2
```

**Note:** % operator can only be used with integers.

Assignment Operators

The most common assignment operator is =. This operator assigns the value in right side to the left side. For example:

var=5 //5 is assigned to var

a=c; //value of c is assigned to a

5=c; // Error! 5 is a constant.

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator	C  = 2 is same as C = C   2

**Relational Operator**

Relational operator's checks relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0.

For example:  $a > b$

Here,  $>$  is a relational operator. If  $a$  is greater than  $b$ ,  $a > b$  returns 1 if not then, it returns 0.

Following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

**Example**

Try the following example to understand all the relational operators available in C programming language:

```
#include <stdio.h>
void main()
{
    int a = 21; int b = 10;
    int c ;
    if( a == b )
    {
        printf("Line 1 - a is equal to b\n");
    }
    else
    {
        printf("Line 1 - a is not equal to b\n");
    }
    if ( a < b )
```



```
{
    printf("Line 2 - a is less than b\n" );
}
else
{
    printf("Line 2 - a is not less than b\n" );
}
if ( a > b )
{
    printf("Line 3 - a is greater than b\n" );
}
else
{
    printf("Line 3 - a is not greater than b\n" );
}
/* Lets change value of a and b */
a = 5; b = 20;
if ( a <= b )
{
    printf("Line 4 - a is either less than or equal to b\n" );
}
if ( b >= a )
{
    printf("Line 5 - b is either greater than or equal to b\n" );
}
}
```

When you compile and execute the above program it produces the following result:

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b
```

नाह ज्ञानन सद्रश  
**SHREE MEDHA DEGRE COLLEGE**

**Logical Operators**

Logical operators are used to combine expressions containing relation operators. In C, there are 3 logical operators:

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

**Example**

Try the following example to understand all the logical operators available in C programming language:

```
#include <stdio.h>
void main()
{
    int a = 5; int b = 20; int c ;
    if ( a && b )
    {
        printf("Line 1 - Condition is true\n");
    }
    if ( a || b )
    {
        printf("Line 2 - Condition is true\n");
    }
    /* lets change the value of a and b */
    a = 0; b = 10;
    if ( a && b )
    {
        printf("Line 3 - Condition is true\n");
    }
    else
    {
        printf("Line 3 - Condition is not true\n");
    }
    if ( !(a && b) )
    {
        printf("Line 4 - Condition is true\n");
    }
}
```

When you compile and execute the above program it produces the following result:

Line 1 - Condition is true  
 Line 2 - Condition is true  
 Line 3 - Condition is not true  
 Line 4 - Condition is true

### Bitwise Operators

Bitwise operators are special types of operators that are used in programming the processor. In processor, mathematical operations like: addition, subtraction, addition and division are done using the bitwise operators which makes processing faster and saves power.

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

P	Q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

***Bitwise AND operator in C programming.***

The output of logical AND is 1 if both the corresponding bits of operand is 1. If either of bit is 0 or both bits are 0, the output will be 0. It is a binary operator (works on two operands) and indicated in C programming by & symbol. Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

**Bit Operation of 12 and 25**

00001100

&00011001

-----  
00001000 = 8 (In decimal)

As, every bitwise operator works on each bit of data. The corresponding bits of two inputs are check and if both bits are 1 then only the output will be 1. In this case, both bits are 1 at only one position, i.e, fourth position from the right, hence the output bit of that position is 1 and all other bits are 0.

```
#include <stdio.h>
void main()
{
    int a=12,b=39;
    printf("Output=%d",a&b);
    getch();
}
```

**Output**

Output=4

***Bitwise OR operator in C***

The output of bitwise OR is 1 if either of the bit is 1 or both the bits are 1. In C Programming, bitwise OR operator is denoted by |.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

**Bitwise OR Operation of 12 and 25**

00001100

| 00011001

-----  
00011101 = 29 (In decimal)

```
#include <stdio.h>
int main()
{
    int a=12,b=25;
    printf("Output=%d",a | b);
    return 0;
}
```

**Output**

Output=29

***C Programming Bitwise XOR (exclusive OR) operator***

The output of bitwise XOR operator is 1 if the corresponding bits of two operators are opposite (i.e., To get corresponding output bit 1; if corresponding bit of first operand is 0 then, corresponding bit of second operand should be 1 and vice-versa.). It is denoted by ^.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

00001100

| 00011001

-----  
00010101 = 21 (In decimal)

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

```
int main()
```

```
{
```

```
    int a=12,b=25;
```

```
    printf("Output=%d",a^b);
```

```
    return 0;
```

```
}
```

Output=21

***Bitwise compliment operator***

Bitwise compliment operator is an unary operator (works on one operand only). It changes the corresponding bit of the operand to opposite bit, i.e., 0 to 1 and 1 to 0. It is denoted by ~.

35=00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

-----  
11011100 = 220 (In decimal)

Twist in bitwise complement operator in C Programming

Output of ~35 shown by compiler won't be 220, instead it shows -36. For any integer  $n$ , bitwise complement of  $n$  will be  $-(n+1)$ . To understand this, you should understand the concept of 2's complement.

***2's Complement***

Two's complement is the operation on binary numbers which allows number to write it in different form. The 2's complement of number is equal to the complement of number plus 1. For example:

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

**Note: Overflow is ignored while computing 2's complement.**



If we consider the bitwise complement of 35, 220(in decimal) is converted into 2's complement which is -36. Thus, the output shown by computer will be -36 instead of 220.

**How is bitwise complement of any number  $N = -(N+1)$ ?**

bitwise complement of  $N = \sim N$  (represented in 2's complement form)

2's complement of  $\sim N = -(\sim(\sim N)+1) = -(N+1)$

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

```
int main()
```

```
{
```

```
    printf("complement=%d\n",~35);
```

```
    printf("complement=%d\n",~-12);
```

```
    return 0;
```

```
}
```

```
complement=-36
```

**Output=11*****Shift Operator in C programming***

There are two shift operators in C programming: Right shift operator and Left shift operator.

**Right Shift Operator**

Right shift operator moves the all bits towards the right by certain number of bits which can be specified. It is denoted by  $\gg$ .

212 = 11010100 (In binary)

212 $\gg$ 2 = 00110101 (In binary) [Right shift by two bits]

212 $\gg$ 7 = 00000001 (In binary)

212 $\gg$ 8 = 00000000

212 $\gg$ 0 = 11010100 (No Shift)

**Left Shift Operator**

Left shift operator moves the all bits towards the left by certain number of bits which can be specified. It is denoted by  $\ll$ .

212 = 11010100 (In binary)

212 $\ll$ 1 = 10101000 (In binary) [Left shift by one bit]

212 $\ll$ 0 = 11010100 (Shift by 0)

212 $\ll$ 4 = 01000000 (In binary) = 3392(In decimal)

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

```
void main()
```

```
{
```

```
    int num=212,i;
```

```
    for (i=0;i<=2;++i)
```

```
        printf("Right shift by %d: %d\n",i,num>>i);
```

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

```
    for (i=0;i<=2;++i)
```

```
        printf("Left shift by %d: %d\n",i,num<<i);
```

```
    getch();
```

```
}
```

Right Shift by 0: 212

Right Shift by 1: 106

Right Shift by 2: 53

Left Shift by 0: 212

Left Shift by 1: 424

Left Shift by 2: 848

Interesting thing to note in Left and Right Shift

For any positive number, right shift is equal to integer division of that number by (shift bit plus one) and for any integer left shift is equal to the multiplication of that number by (shift bit plus one)

### ***Conditional operators (?:) or Ternary Operator***

Conditional operators are used in decision making in C programming, i.e., executes different statements according to test condition whether it is either true or false.

- They are also called as Ternary Operator.
- They also called as ?: operator
- Ternary Operators takes on 3 Arguments

#### **Syntax of conditional operator**



where

- **expression1** is Condition
- **expression2** is Statement Followed if Condition is True
- **expression2** is Statement Followed if Condition is False

#### **Meaning of Syntax:**

- Expression1 is nothing but **Boolean Condition** i.e it results into either **TRUE** or **FALSE**
- If result of expression1 is **TRUE** then expression2 is Executed
- Expression1 is said to be **TRUE** if its result is NON-ZERO
- If result of expression1 is **FALSE** then expression3 is Executed
- Expression1 is said to be **FALSE** if its result is ZERO

**Example of conditional operator**

```
#include <stdio.h>
void main()
{
    char feb;
    int days;
    printf("Enter 1 if the year is leap year otherwise enter 0: ");
    scanf("%c",&feb);
    days=(feb=='1')?29:28;
    /*If test condition (feb=='1') is true, days will be equal to 29. */
    /*If test condition (feb=='1') is false, days will be equal to 28. */
    printf("Number of days in February = %d",days);
    getch();
}
```

**OUTPUT**

Enter 1 if the year is leap year otherwise enter n: 1  
 Number of days in February = 29

Other operators such as &(reference operator), \*(dereference operator) and ->(member selection) operator will be discussed in pointer chapter.

***Special or Other Operators***

There are few other important operators including sizeof and ? : supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of an variable.	&a; will give actaul address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

**Comma Operator**

Comma operators are used to link related expressions together. For example:  
 int a,c=5,d;

**The sizeof operator**

It is a unary operator which is used in finding the size of data type, constant, arrays, structure etc. For example:

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

```

double c;
char d;
printf("Size of int=%d bytes\n",sizeof(a));
printf("Size of float=%d bytes\n",sizeof(b));
printf("Size of double=%d bytes\n",sizeof(c));
printf("Size of char=%d byte\n",sizeof(d));
return 0;
getch();
}

```

**Output**

```

Size of int=4 bytes
Size of float=4 bytes
Size of double=8 bytes
Size of char=1 byte

```

**Expressions**

**Operands** are variables or expressions which are used in **operators** to evaluate the expression.

**Combination of operands and operators form an Expression.** Statements like  $a = b + 3$ ,  $++z$  and  $300 > (8 * k)$  are all expressions.

Every expression consists of at least one *operand* and can have one or more *operators*. Operands are values, whereas operators are symbols that represent particular actions. In the expression

$x + 5$

**$x$  and  $5$  are operands, and  $+$  is an operator.**

Expressions are used in programming languages, database systems, and spreadsheet applications.

***Arithmetic Expressions***

Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax. C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax. Some examples of mathematical expressions written in proper syntax of C are:

Algebraic expression	C expression
$axb-c$	$a*b-c$
$(m+n)(x+y)$	$(m+n)*(x+y)$
$\left[ \frac{ab}{c} \right]$	$a*b/c$
$3x^2+2x+1$	$3*x*x+2*x+1$
$\frac{a}{b}$	$a/b$
$S = \frac{a+b+c}{2}$	$S=(a+b+c)/2$

$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$	<code>area=sqrt(s*(s-a)*(s-b)*(s-c))</code>
$\sin\left(\frac{b}{\sqrt{a^2+b^2}}\right)$	<code>sin(b/sqrt(a*a+b*b))</code>
$\tau_1 = \sqrt{\left\{\frac{\sigma_x - \sigma_y}{2}\right\}^2 + \tau_{xy}^2}$	<code>tow1=sqrt((rowx-rowy)/2+tow*x*y*y)</code>
$\tau_1 = \sqrt{\left\{\frac{\sigma_x - \sigma_y}{2}\right\}^2 + \tau_{xy}^2}$	<code>tow1=sqrt(pow((rowx-rowy)/2,2)+tow*x*y*y)</code>
$y = \frac{\alpha + \beta}{\sin \theta} +  x $	<code>y=(alpha+beta)/sin(theta*3.1416/180)+abs(x)</code>

### *Types of Expressions*

- 1) **Arithmetic expressions** - expressions which contain operands and arithmetic operators
  - a) **Integer expressions** - expressions which contains integers and operators
  - b) **Real expressions** - expressions which contains floating point values and operators
  - c) **Mixed mode arithmetic expressions** - expressions which contain both integer and real operands
- 2) **Relational expressions** - expressions which contain relational operators and operands
- 3) **Logical expressions** - expressions which contain logical operators and operands
- 4) **Assignment expressions** and so on... - expressions which contain assignment operators and operands

### Expression Evaluation

Expressions are evaluated using an assignment statement of the form:

**variable = expression**

In the above syntax, **variable** is any valid C variable name. When the statement like the above form is encountered, the expression is evaluated first and then the value is assigned to the variable on the left hand side. All variables used in the expression must be declared and assigned values before evaluation is attempted. Examples of expressions are:

**x = a\*b-c**  
**y=b/c\*a**  
**z=a-b/c+d**

Expressions are evaluated based on operator precedence and associativity rules when an expression contains more than one operator.



### Type Conversion/Casting in Expressions

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the cast operator.

New data type should be mentioned before the variable name or value in brackets which to be typecast.

#### **Why do we need type casting?**

*Type casting would be pretty handy when your inputs to an operation is of a particular type and you need more precision in the output. Let's see the use of type casting through a simple example*

```
#include <stdio.h>
int main()
{
    int value1 = 10, value2 = 3;
    float result;
    result = value1 / value2;
    printf("Result without type casting: %f", result);
    return 0;
}
```

#### **Output:**

Result without type casting: 3.000000

In the above example we assign the result of division operation to float type variable result. The result is supposed to be 3.333333. But we get the result as 3.000000. *Can you think why?* Because the two operands of the expression, *value1* and *value2* are of int type. This causes compiler to treat the result also as integer; that means, it will ignore the decimal part of the result and store the integer result in float variable.

*So how do you get the result with correct precision?* Just cast the type of expression to the type you need! You can see the solution below.

```
#include <stdio.h>
void main()
{
    int value1 = 10, value2 = 3;
    float result;
    result = (float) value1 / value2;
    printf("Result without type casting: %f", result);
    getch();
}
```

#### **Output:**

Result without type casting: 3.333333

*One important thing to notice here is that, type casting has more precedence than the divide operation (/). At first, value1 will be converted to float type and then division operation is performed on the float variable.*

### WHAT IS TYPE CASTING IN C LANGUAGE?

Converting an expression of a given type into another type is known as type-casting. typecasting is more use in c language programming.

Here, it is best practice to convert lower data type to higher data type to avoid data loss.

Data will be truncated when higher data type is converted to lower. For example, if float is converted to int, data which is present after decimal point will be lost.

There are two types of type casting in c language.

#### **Implicit and Explicit Type Casting**

Type casting can either be performed by the compiler automatically or user can specify them using the syntax described above. Former type is called implicit type casting and the latter is called explicit type casting.

#### **What is implicit type casting/ implicit type conversion?**

We've explained the explicit type conversion in the above paragraphs. Now let's take a look at how implicit type conversion works. In implicit type conversion compiler takes care of the casting without requiring the programmer to specify them explicitly. For example, in the program given below, compiler converts the sum of two float numbers to an int.

```
#include <stdio.h>
void main()
{
    float value1 = 2.2;
    float value2 = 3.3;
    int result;
    result = value1 + value2;
    printf("Result : %d", result);
    getch();}
```

#### **Output:**

Result: 5

**Note:** Though compiler performs implicit type casting, it's always recommended to specify the conversion explicitly because this improves the code readability and makes it easier to port to other platform.

#### **Rules for Implicit Type Casting**

When different operands of an expression are of different type, compiler performs implicit type conversion to match them. Below are the rules followed by compiler to perform the implicit conversion.

- Integer Promotion: All types lower than int (char, short etc) are first promoted to int.
- If the types of operands differ even after integer promotion, then following actions are taken
- If one of the operand is long double, convert all others to long double
- If one of the operand is double, convert all others to double
- If one of the operand is float, convert all others to float

- If one of the operand is float, convert all others to float

... This operation proceeds from the highest type to lowest.

Order of the data types from highest to lowest is given below

long double > double > float > unsigned long long > long long > unsigned long > long > unsigned int > int

### **EXPLICIT CONVERSION**

In c language, many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion.

They are not automatically performed when a value is copied to a compatible type in program.

#### **Example: -**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=20;
    short p;
    clrscr();
    p = (short) i; // Explicit conversion
    printf("Explicit value is %d",p);
    getch();
}
```

#### **Output**

Explicit value is 20.

#### **Upcasting and Downcasting**

Upcast means converting *lower* (like int) to *higher* (float, long int) data type. The reverse is called downcast.

Upcast results in no information loss. But downcast results in information loss as lower data type have lesser bits and can hold the lesser amount of information/data. One data type considers *higher* if maximum value allowed to store in it is greater than the other data type. For example, float is lower compared to double because double can store more precisions

#### **C Type Casting with examples for Float to Int/Int to Char**

### **Operator Precedence and Associativity**

Every C operator has a precedence (priority) associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are different levels of operator precedence and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators in the same level of precedence are evaluated from left to right or from right to left, based on the associativity property of an operator. In the below table we can look at the precedence levels of operators

and also the associativity of the operators within the same level. Rank 1 indicates the highest precedence and Rank 9 indicates lowest precedence. Priority

Operator	Priority	Associativity
{}, (), []	1	Left to right
++, --, !	2	Right to left
*, /, %	3	Left to right
+, -	4	Left to right
<, <=, >, >=, ==, !=	5	Left to right
&&	6	Left to right
	7	Left to right
?:	8	Right to left
=, +=, -=, *=, /=, %/=	9	Right to left

Suppose, we have an arithmetic expression as:

$$x = 9 - 12 / 3 + 3 * 2 - 1$$

This expression is evaluated in two left to right passes as:

#### First Pass

Step 1:  $x = 9 - 4 + 3 * 2 - 1$

Step 2:  $x = 9 - 4 + 6 - 1$

#### Second Pass

Step 1:  $x = 5 + 6 - 1$

Step 2:  $x = 11 - 1$

Step 3:  $x = 10$

But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example,

$$x = 9 - 12 / (3 + 3) * (2 - 1)$$

When parentheses are present then the expression inside the parenthesis are evaluated first from left to right. The expression is now evaluated in three passes as:

#### First Pass

Step 1:  $x = 9 - 12 / 6 * (2 - 1)$

Step 2:  $x = 9 - 12 / 6 * 1$

#### Second Pass

Step 1:  $x = 9 - 2 * 1$

Step 2:  $x = 9 - 2$

#### Third Pass

Step 3:  $x = 7$

There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.

**For example, we have an expression as:**

$$x = 9 - ((12 / 3) + 3 * 2) - 1$$

**The expression is now evaluated as:**

**First Pass:**

$$\text{Step 1: } x = 9 - (4 + 3 * 2) - 1$$

$$\text{Step 2: } x = 9 - (4 + 6) - 1$$

$$\text{Step 3: } x = 9 - 10 - 1$$

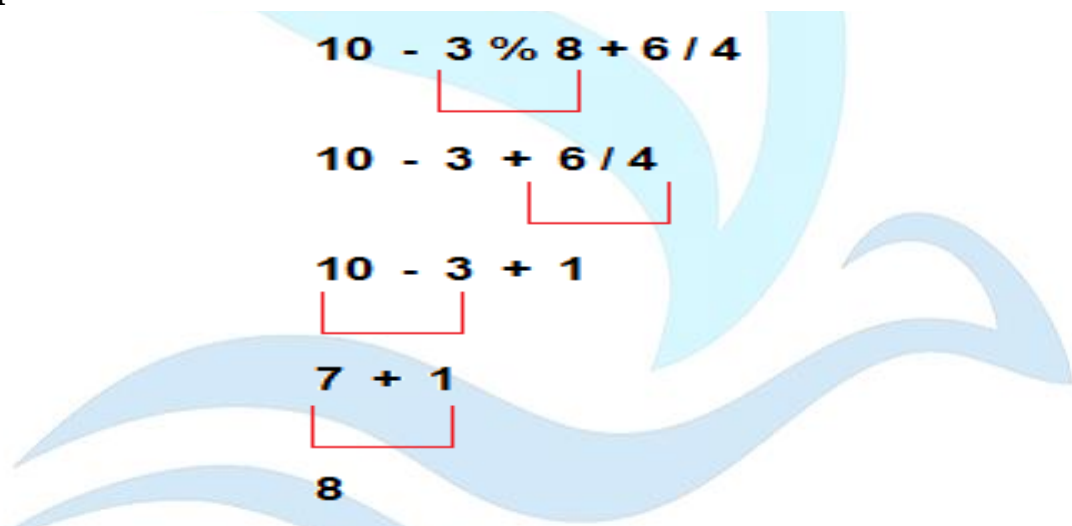
**Second Pass**

$$\text{Step 1: } x = -1 - 1$$

$$\text{Step 2: } x = -2$$

Note: The number of evaluation steps is equal to the number of operators in the arithmetic expression.

**Example 1:**



**Example 2: where a=5**

$$\begin{aligned} &17 - 8 / 4 * 2 + 3 - ++a \\ &17 - 8 / 4 * 2 + 3 - 6 \\ &17 - 2 * 2 + 3 - 6 \\ &17 - 4 + 3 - 6 \\ &13 + 4 - 6 \\ &16 - 6 \\ &10 \end{aligned}$$



**Mathematical Functions**

There are various standard library functions and a macro defined under math.h to perform mathematical operations in C programming.

Library functions under "math.h"	
Function	Work of function
<u>acos</u> ( )	Computes arc cosine of the argument
<u>acosh</u> ( )	Computes hyperbolic arc cosine of the argument
<u>asin</u> ( )	Computes arc sine of the argument
<u>asinh</u> ( )	Computes hyperbolic arc sine of the argument
<u>atan</u> ( )	Computes arc tangent of the argument
<u>atanh</u> ( )	Computes hyperbolic arc tangent of the argument
<u>atan2</u> ( )	Computes arc tangent and determine the quadrant using sign
<u>cbrt</u> ( )	Computes cube root of the argument
<u>ceil</u> ( )	Returns nearest integer greater than argument passed
<u>cos</u> ( )	Computes the cosine of the argument
<u>cosh</u> ( )	Computes the hyperbolic cosine of the argument
<u>exp</u> ( )	Computes the e raised to given power
<u>fabs</u> ( )	Computes absolute argument of floating point argument
<u>floor</u> ( )	Returns nearest integer lower than the argument passed.
<u>hypot</u> ( )	Computes square root of sum of two arguments (Computes hypotenuse)
<u>log</u> ( )	Computes natural logarithm
<u>log10</u> ( )	Computes logarithm of base argument 10
<u>pow</u> ( )	Computes the number raised to given power
<u>sin</u> ( )	Computes sine of the argument
<u>sinh</u> ( )	Computes hyperbolic sine of the argument

## Library functions under "math.h"

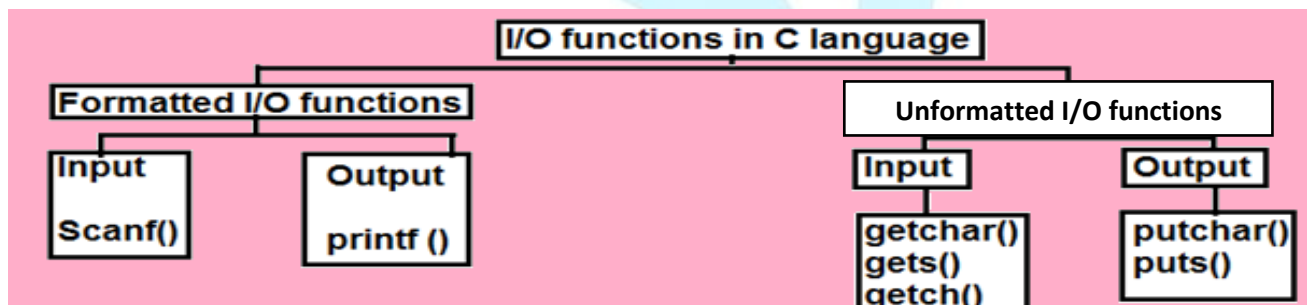
Function	Work of function
<u>sqrt()</u>	Computes square root of the argument
<u>tan()</u>	Computes tangent of the argument
<u>tanh()</u>	Computes hyperbolic tangent of the argument

**Managing Input & Output Operations**

In any programming language, the interface forms a very important part. It deals with taking data from the user and displaying back the output.

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

**Unformatted Input Output Functions**

Unformatted input and output functions are only work with character data type. Unformatted input and output functions do not require any format specifiers. Because they only work with character data type.

**Reading A Character or Unformatted Input Function**

The simplest of all input/output operations is reading a character from the standard input unit (usually the keyboard) and writing it to the standard output unit (usually the screen). Reading a single character can be done by using the function **getchar**. The **getchar** takes the following form:

```
variable_name = getchar();
```

variable\_name is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the

right-hand side of an assignment statement, the character value of **getchar** is in turn assigned to the variable name on the left.

**For example**

char name;

**name = getchar();**

will assign the character 'H' to the variable name when we press the key H on the keyboard. Since getchar is a function, it requires a set of parentheses as shown.

**Example program:**

```
#include <stdio.h> // assigns stdio-h header file to your program
void main () // Indicates the starting point of the program.
{
char C, // variable declaration
printf ("Type one character:"); // message to user
C = getchar () ; // get a character from key board and Stores it in variable C.
printf (" The character you typed is = %c", C) ; // output
} // Statement which displays value of C on
// Standard screen.
```

C supports testing of the character keyed in by the user by including the file ctype.h & the functions which can be used after including this file are

isalnum(c) Is c an alphanumeric character?

isalpha(c) Is c an alphabetic character?

isdigit(c) Is c a digit?

islower(c) Is c a lower case character?

isprint(c) Is c a printable character?

ispunct(c) Is c a punctuation mark?

isspace(c) Is c a white space character?

isupper(c) Is c a upper case character?

**Writing a character or Unformatted Output Function**

The simplest of all output operations writing it to the standard output unit (usually the screen). Writing a single character can be done by using the function **putchar**. The putchar takes the following form:

**putchar (variable name);**

Where variable is a valid C type variable that has already been declared Ex:-

putchar ( );

**Displays the value stored in variable C to the standard screen.**

**Program shows the use of getchar function in an interactive environment.**

```
#include <stdio.h> // Inserts stdio.h header file into the Pgm
void main () // Beginning of main function.
```

```
{
char in; // character declaration of variable in.
printf (" please enter one character"); // message to user
in = getchar ( ) ; // assign the keyboard input value to in.
putchar (in); // output 'in' value to standard screen.
}
```

### **String input and output:**

The **gets** function retrieves the string from standard input device while **put S** outputs the string to the standard output device. A string is an array or set of characters.

The function **gets** accepts the name of the string as a parameter, and fills the string with characters that are input from the keyboard till newline character is encountered. (That is till we press the enter key). At the end the function **gets** appends a null terminator as must be done to any string and returns.

**The puts function displays the contents stored in its parameter on the standard screen.**

**The standard form of the gets function is**

**gets (str)**

Here "str" is a string variable.

**NOTE:** It is recommended NOT to make use of **gets()** function. It is unsafe and can even crash the C program itself. Above description is for theoretical illustration purpose only.

**The standard form for the puts character is**

**puts (str)**

Where str is a string variable.

**Example program (Involving both gets and puts)**

```
# include < stdio.h >
void main ( )
{
char s [80];
printf ("Type a string less than 80 characters:");
gets (s);
printf ("The string types is:");
puts(s);
}
```

**Formatted I/O functions: -**

The formatted I/O functions read & write all types of data values. They require conversion symbol to identify the data type.

**Formatted input statement or functions**

**scanf(\_)**

This is formatted input function which can read data from the standard input device into the variable in different formats by the user.

**The function is used to read data more than one variable at a time.**

**Syntax:-**

<b>scanf("control string", &amp;arg);</b>
---

**Ex:-**

**scanf("%d", &no);**

1. It reads all types of data values.
2. It is used for run time assignment of variables.
3. It requires conversion symbol to identify the data to be read by the program during execution.
4. The ampersand (&) symbol is used to indicate the memory location of the variable. So that the value read would be placed at that location.
5. The control string may contain format code, escape sequence & width specifier.
6. The format code specifies the type of input data.
7. For each variable there must be a format code.

**Syntax:**

<b>scanf("control string" &amp;arg1, &amp;arg2, . . . ,&amp;argn);</b>
--

The control string specifies the field format in which the data is to be entered & the args arg1,.... arg n specify the addition of the locations where the data is stored.

Control string & arguments are separated by commas.

Control string contains field specifications, which direct the interpretation of i/p data. It may include:

1. Field (or format) specifications, consisting of the conversion character %, a data type character and an optional number specifying the field width.
2. Blanks, tabs, or new lines.

The data character indicates the type of data that is to be assigned to the variable associated with the corresponding argument.

The field width specifier is optional.

**Commonly used scanf ( ) FORMAT CODES are:**

Code or Control Strings	Meaning
%c	Read a single character
%d	Read decimal integer
%f	Read a floating point value
%e	Read a floating point value
%O	Read a octal integer
%s	Read a string
%u	Read an unsigned decimal integer
%x	Read a hexadecimal integer
%[""]	Read a string of word(s)

### **Inputting integer numbers: -**

The field specification for reading a integer is:

**Syntax: - %wd**

The % sign indicates that a conversion specification follows. W is a integer number that specifies the field width of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode.

**Ex:- scanf("%2d%5d", &no1, &no2);**

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

**Ex:- scanf("%d%\*d%d", &a, &b);**

Will assign the data 123 456 789

Now 123 is assigned to 'a' 456 is skipped (because of \*) 789 is to 'b'.

### **Inputting real numbers: -**

The field width of real numbers is not to be specified and therefore scanf reads real numbers using the simple specification %f for both the notification.

**Ex:- scanf("%f%f%f", &x, &y, &z);**

With the input data 475.1 43.21-1 678

Will assign the values:475.1 to x

4.321 to y and

678.0 to z

### **Inputting character strings:**

It is done in 2 ways, by using getchar function and by using scanf function. A scanf function can input strings containing more than one character.

**Syntax: %ws or %wc**

The corresponding argument should be a pointer to a character array.

%c is used to read a **single character**.

%s is used to read **words or paragraphs**.

### **Reading mixed data types: -**

It is possible to use one scanf statement to input a data line containing mixed mode

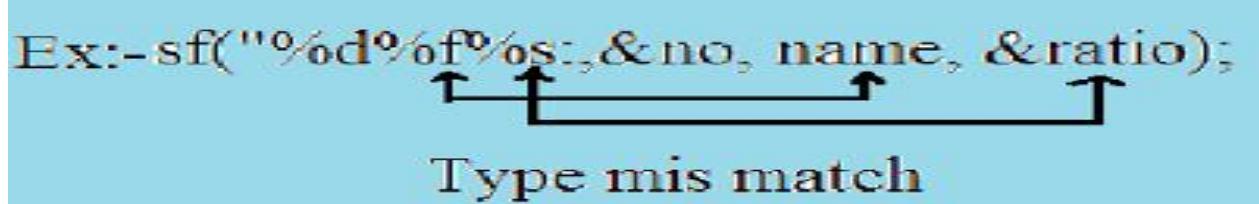


data. in such cases, care should be exercised to ensure that the i/p data items match the control specifications in order and type.

**Ex:-** `scanf("%d%c%f%s", &no, &code, &ration, name);`

### **Detection of errors in input:**

Only one error is encountered while reading i/p i.e mismatch of format specifier & arguments.



The diagram shows a code snippet: `Ex:- sf("%d%f%s;,&no, name, &ratio);`. Below the code, four arrows point upwards to the format specifiers: the first arrow points to `%d`, the second to `%f`, the third to `%s`, and the fourth to the semicolon `;`. Below these arrows, the text "Type mis match" is written, indicating that the format specifiers do not match the variable types or the statement is syntactically incorrect.

### **Rules while using scanf: -**

1. Every variable to be read must have field specification.
2. For each field specification there must be variable address of proper type.
3. Never end the format string with white space. It is a fatal error.
4. Scanf read until.

A white space character is formed.

The maximum no of characters has been read.

An error is detected.

The end of file is reached

### **Formatted output:**

**printf():-**

printf function is used to print the data (or) more than one data item in different formats on the standard o/p device.

**Syntax:-**

**`printf("control string", arg1, arg2,...,argn);`**

**Control string consists of 3 types of items:**

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the o/p format for display of each item.
3. Escape sequence characters such as `\n`, `\t` & `\b`.

The control string indicates how many arguments follow & printed according to the specifications of the control string.

The arguments should match in number, order and type with format specifications.

### **Output of integer numbers:**

The format specification for printf on integer number is:

**%wd**

Where **w** specifies the minimum field width for the o/p **d** specifies that the value to be printed is an integer.

The number is written right-justified in the given field with:

**Ex:-** printf("%6d",9876);

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

it is possible to force the printing to be left-justified by placing a minus sign directly after the %character.

**Ex:-** printf("%-6d",9876);

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

it is also possible to pad with zero's the leading blanks by placing a 0 before the field width specifier.

**Ex:-** printf("%06d",9876);

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

The minus (-) & zero are known as flags.

**Ex:-** printf("%d",9876);

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

### **Output of real numbers: -**

The output of a real no may be displayed in decimal notation using the following format specification

**%w.pf**

The integer **w** indication the minimum number of positions that are to be used for the display of the value and the integer **p** indicates the number of digits to be displayed after the decimal point (precision) **f** format specifies.

We can also display a real no in exponential notation by using the specification.

**%w.pe**

**Ex:-** y=98.76546  
printf("%7.4f", y);  
printf("%7.2f",y);  
printf("%-7.2f", y);

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

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

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

### **Printing of a single character:**

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

**%wc**

The character will be displayed right justified in the field of w columns. We can make the display left-justified by placing a minus sign before the integer w. The default value for w is 1.

### **Printing of strings:**

The format specification for outputting strings is similar to real numbers

**%w.ps**

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

**Ex:**

**%20s**

				N	E	W		D	E	L	H	I		1	1	0	0
--	--	--	--	---	---	---	--	---	---	---	---	---	--	---	---	---	---

**%20.10s**

N	E	W		D	E	L	H	I									
---	---	---	--	---	---	---	---	---	--	--	--	--	--	--	--	--	--

### **Mixed data output: -**

printf uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order and type.

If there are not enough variables or if they are of the wrong type, the o/p results will be incorrect.

**Ex:- printf("%d%f%s%c",a, b, c, d);**

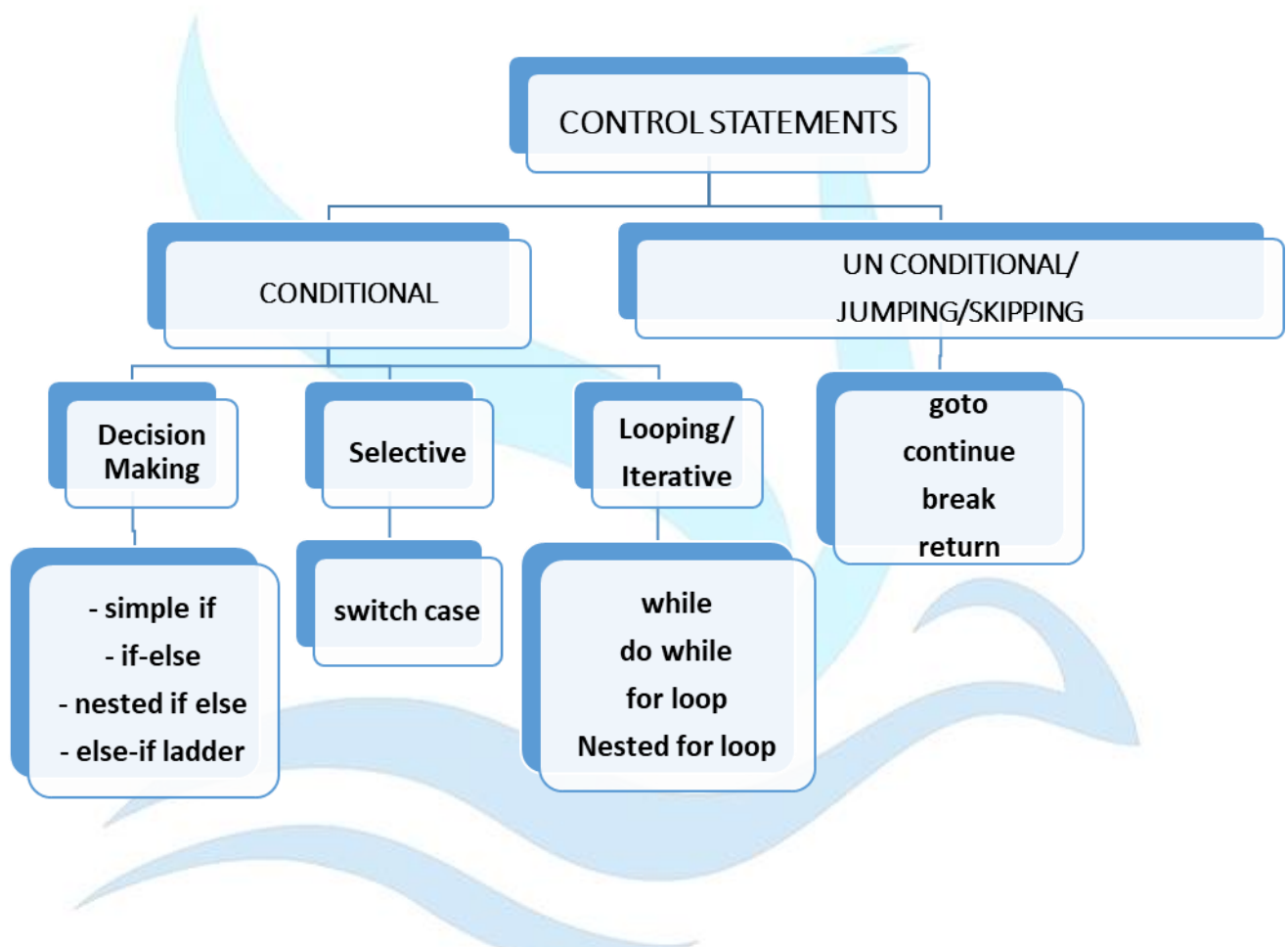


नहि ज्ञानेन सद्रश्  
SHREE MEDHA DEGRE COLLEGE

### Control statements

**Control statements** enable us to specify the flow of program control; i.e., the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

### Types of Control Statements



### CONDITIONAL CONTROL STATEMENTS:

- Conditional Control Statements involves performing a Logic Test.
- This Test results either a TRUE or FALSE.

Depending upon the truthness or falsity of the condition, the statements to be executed is determined. After that, the control transfers to the statement in the program and start executing the statements. This is known as Conditional Execution.

### Decision Making or Branching Control Statements

Decision-making statements are the statements that are used to verify a given condition and decides whether a block of statements gets executed or not based on the condition result.

In C four conditional control statements are widely used:

- ❖ If-statement
- ❖ If-else statement
- ❖ Nested-if-else statement
- ❖ else-if ladder

### Simple if statement or One way Branching

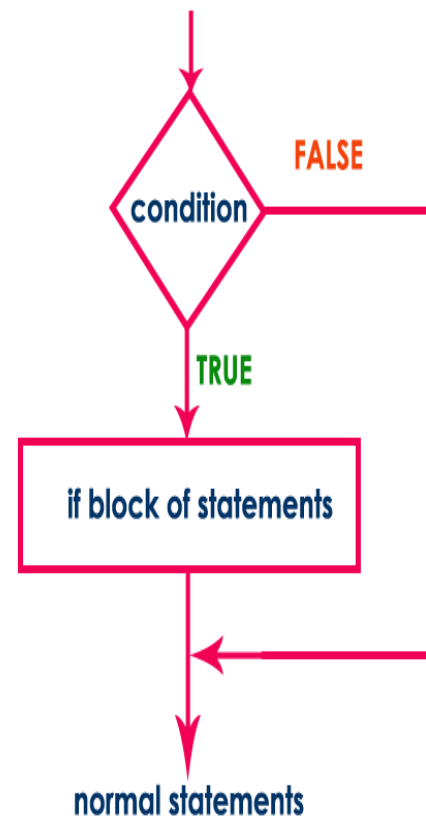
Simple if statement is used to verify the given condition and executes the block of statements based on the condition result. The simple if statement evaluates specified condition. If it is TRUE, it executes the next statement or block of statements. If the condition is FALSE, it skips the execution of the next statement or block of statements.

The general syntax and execution flow of the simple if statement is as follows...

#### Syntax

```
if ( condition )  
{  
    ...  
    block of statements;  
    ...  
}
```

#### Execution flow diagram



Where

**Condition:** is a logical expression that results in TRUE or FALSE.

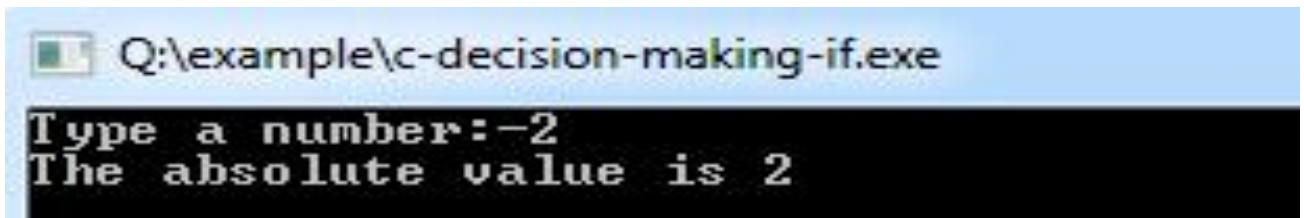
**Statement:** a simple statement (single statement) or compound statement (collection of two or more statement).

**Explanation:** If the logical condition is TRUE then statement1 is executed and If the logical condition is FALSE then control transfers to the next executable statement. Simple if statement is used when we have only one option that is executed or skipped based on a condition.

**//Program to convert negative value to absolute using simple if**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int number;
    printf( " Type a number: ");
    scanf( "%d", &number);
    // check whether the number is negative number
    if (number < 0)
    {
        // If it is a negative then convert it into positive.
        number = -number;
        printf( " The absolute value is % d\n ", number);
    }
    getch();
}
```

**OUTPUT**



```
Q:\example\c-decision-making-if.exe
Type a number:-2
The absolute value is 2
```



नहि ज्ञानेन सद्रशं  
SHREE MEDHA DEGREE COLLEGE



**if - else statement OR Two Way Branching**

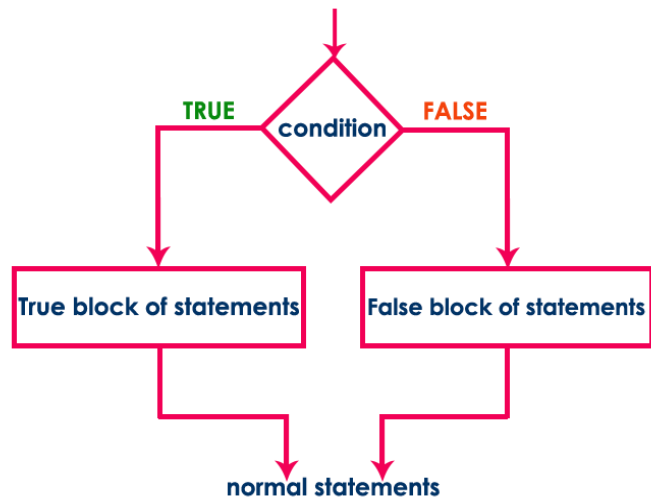
The if - else statement is used to verify the given condition and executes only one out of the two blocks of statements based on the condition result. The if-else statement evaluates the specified condition. If it is TRUE, it executes a block of statements (True block). If the condition is FALSE, it executes another block of statements (False block). The general syntax and execution flow of the if-else statement is as follows...

**Syntax**

```

if ( condition )
{
    ....
    True block of statements;
    ....
}
else
{
    ....
    False block of statements;
    ....
}

```

**Execution flow diagram**

Where

**Condition:** is a logical expression that results in TRUE or FALSE.

**Statement:** a simple statement (single statement) or compound statement (collection of two or more statement).

**Explanation:** If the condition specified in the if statement evaluates to true, the statements inside the if-block are executed and then the control gets transferred to the statement immediately after the if-block. Even if the condition is false and no else-block is present, control gets transferred to the statement immediately after the if-block.

The else part is required only if a certain sequence of instructions needs to be executed if the condition evaluates to false. It is important to note that the condition is always specified in parentheses and that it is a good practice to enclose the statements in the if block or in the else-block in braces, whether it is a single statement or a compound statement.

The if-else statement is used when we have two options and only one option has to be executed based on a condition result (TRUE or FALSE).

**//Program to find the largest of two numbers using if-else statement**

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b;
    clrscr();
    printf("Enter the values of a & b\n");
    scanf("%d%d",&a,&b);
}

```

```
if(a>b)
printf("%d is larger than %d\n",a,b);
else
printf("(%d is larger than %d\n",b,a);
getch();
}
```

### **OUTPUT**

#### **Run 1:**

Enter the values of a & b  
25  
15  
25 is larger than 15.

#### **Run 2:**

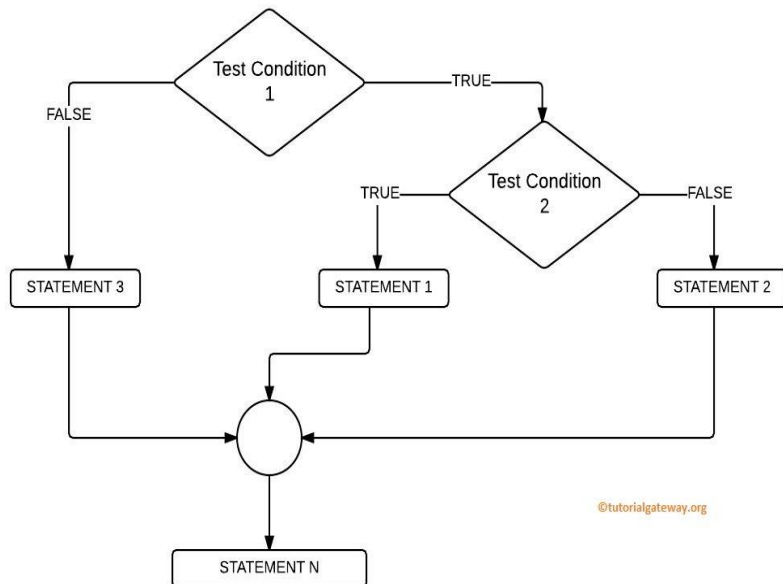
Enter the values of a & b  
35  
45  
45 is larger than 35.

### **Nested If-Else Statement:**

Writing a if statement inside another if statement is called nested if statement. It is used if there are more than two alternatives to select. The nested if statement can be defined using any combination of simple if & if-else statements.

**The syntax of nested-if statement is**

```
if( condition1 )
{
    if( condition2 )
    {
        statement block1;
    }
    else
    {
        statement block2;
    }
}
else
{
    statement block3;
}
```



Where

**Condition:** is a logical expression that results in TRUE or FALSE.

**Statement:** a simple statement (single statement) or compound statement (collection of two or more statement).

**Explanation:** if 'condition1' is false the 'statement-block3' will be executed, otherwise it continues to perform the test for 'condition2'. If the 'condition2' is true the 'statement-block1' is executed otherwise 'statement-block2' is executed.

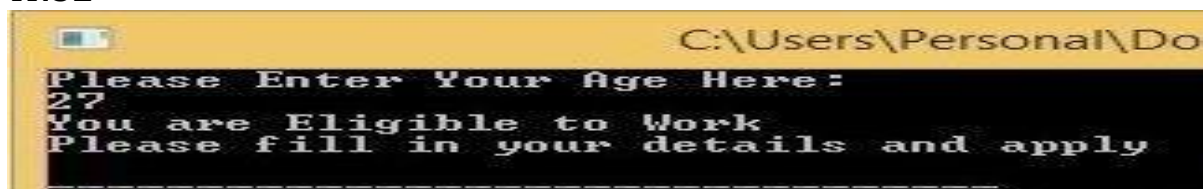
**// Example for Nested If in C Programming**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int age;
    printf("Please Enter Your Age Here:\n");
    scanf("%d",&age);
    if ( age < 18 )
    {
        printf("You are Minor.\n");
        printf("Not Eligible to Work");
    }
    else
    {
        if (age >= 18 && age <= 60 )
        {
            printf("You are Eligible to Work \n");
            printf("Please fill in your details and apply\n");
        }
        else
        {
            printf("You are too old to work as per the Government rules\n");
            printf("Please Collect your pension! \n");
        }
    }
    getch();
}
```

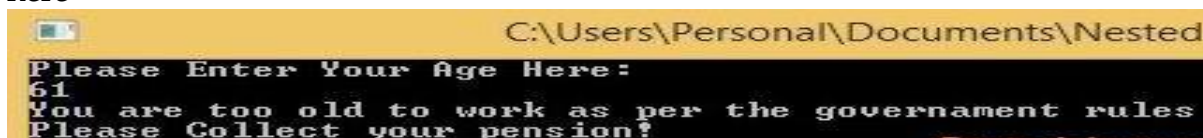
**OUTPUT**



**OUTPUT 2:** Enter the age 27. First If condition is FALSE. Nested IF condition is TRUE



**OUTPUT 3:** Enter the age 61. Both If condition and also Nested IF condition failed here

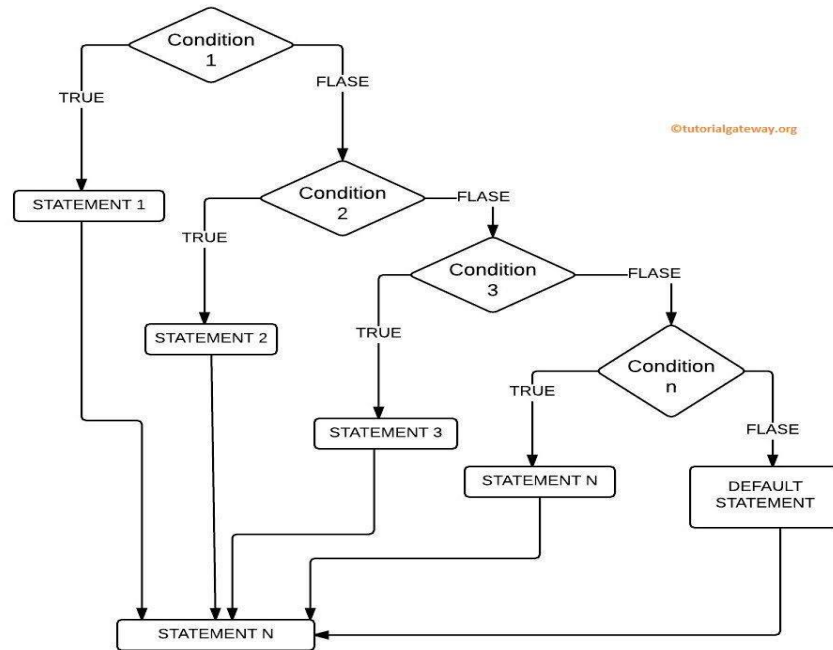


**else-if ladder**

The Else If Statement in C is very useful when we have to check several conditions. We can also use the Nested If statement to achieve the same but as the number of conditions increase, code complexity will also increase. Let us see the syntax of the Else if statement:

**The general format of a nested if-else statement is:**

```
if(condition1)
{
// statement(s);
}
else if(condition2)
{
// statement(s);
}
.
.
else if (conditionN)
{
// statement(s);
}
else
{
// statement(s);
}
```



Else If statement handle multiple statements effectively by executing them sequentially. It will check for the first condition, if the condition is TRUE then it will execute the statements present in that block. If the condition is FALSE then it will check the Next one (Else If condition) and so on.

*There will be some situations where condition 1, condition 2 is TRUE, for example:*

$x = 20, y = 10$

Condition 1:  $x > y$  // TRUE

Condition 2:  $x \neq y$  // TRUE

In these situations, statements under the Condition 1 will be executed because ELSE IF conditions will only be executed if its previous IF or ELSE IF statement fails.

**//Program to find the grade of Student using Else If ladder statement.**

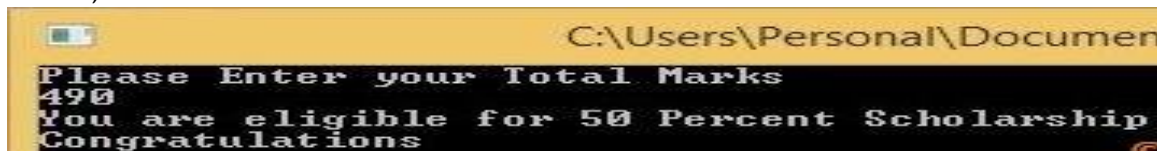
```
#include <stdio.h>
void main()
{
int Totalmarks;
// Imagine you have 6 subjects and Grand total is 600
printf("Please Enter your Total Marks\n");
scanf("%d", &Totalmarks);
if (Totalmarks >= 540)
{
printf("You are eligible for Full Scholarship\n");
printf("Congratulations\n");
}
else if (Totalmarks >= 480)
```

```
{
    printf("You are eligible for 50 Percent Scholarship\n");
    printf("Congratulations\n");
}
else if (Totalmarks >= 400)
{
    printf("You are eligible for 10 Percent Scholarship\n");
    printf("Congratulations\n");
}
else
{
    printf("You are Not eligible for Scholarship\n");
    printf("We are really Sorry for You\n");
}
getch(); }
```

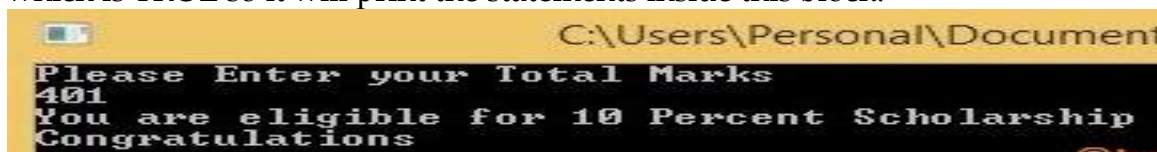
**OUTPUT 1:** We are going to enter Totalmarks = 570. Here first If condition is TRUE



**OUTPUT 2:** We are going to enter Totalmarks = 490 means first IF condition is FALSE. It will check the else if (Totalmarks >= 480), which is TRUE so it will print the statements inside this block. Although else if (Totalmarks >= 400) condition is TRUE, but it won't check this condition.



**OUTPUT 3:** We are going to enter Totalmarks = 401 means first IF condition, else if (Totalmarks >= 480) are FALSE. So, It will check the else if (Totalmarks >= 401), which is TRUE so it will print the statements inside this block.



**OUTPUT 4:** We are going to enter Totalmarks = 380 means all the IF conditions Fail. So, It will print the statements inside the else block.



**Note:** When we use conditional control statement like if statement, condition might be an expression evaluated to a numerical value, a variable or a direct numerical value. If the expression value or direct value is zero the condition becomes FALSE otherwise becomes TRUE.

To understand more consider the following statements

- if(10) - is TRUE
- if(x) - is FALSE if x value is zero otherwise TRUE
- if(a+b) - is FALSE if a+b value is zero otherwise TRUE
- if(a = 99) - is TRUE because a value is non-zero
- if(10, 5, 0) - is FALSE because it considers last value
- if(0) - is FALSE
- if(a=10, b=15, c=0) - is FALSE because last value is zero

## SELECT STATEMENT - SWITCH CASE

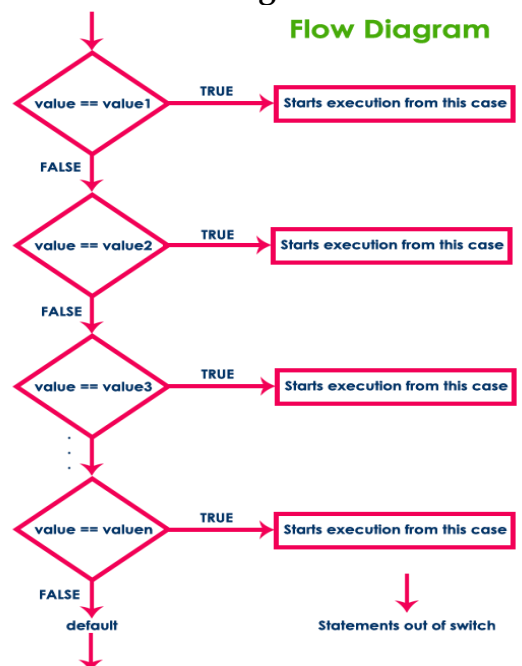
Consider a situation in which we have more number of options out of which we need to select only one option that is to be executed. Such kind of problems can be solved using nested if statement. But as the number of options increases, the complexity of the program also gets increased. This type of problems can be solved very easily using switch statement. Using switch statement, one can select only one option from more number of options very easily. In switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches with the value associated with an option, the execution starts from that option. In **switch statement**, every option is defined as a case.

The switch statement has the following syntax and execution flow diagram...

### Syntax

```
switch ( expression or value )
{
    case value1: set of statements;
    ....
    case value2: set of statements;
    ....
    case value3: set of statements;
    ....
    case value4: set of statements;
    ....
    case value5: set of statements;
    ....
    .
    .
    default: set of statements;
}
```

### Flow Diagram



The switch statement contains one or more number of cases and each case has a value associated with it. At first switch statement, compares the first case value with the switchValue, if it gets matched the execution starts from the first case. If it does not match the switch statement compares the second case value with the switchValue and if it is matched the execution starts from the second case. This process continues until it finds a match. If no case value matches with the switchValue specified in the switch statement, then a special case called **default** is executed.



When a case value matches with the switchValue, the execution starts from that particular case. This execution flow continues with next case statements also. To avoid this, we use "**break**" statement at the end of each case. That means the **break** statement is used to terminate the switch statement. However, it is optional.

**//Program to check whether the given character is vowel or not using switch case.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char ch;
    clrscr();
    printf("Enter any character\n");
    scanf("%c",&ch);
    switch(ch)
    {
        case 'A':
        case 'a':
        case 'E':
        case 'e':
        case 'I':
        case 'i':
        case 'O':
        case 'o':
        case 'U':
        case 'u':printf("%c is a vowel\n",ch);
                break;
        default :printf("%c is not a vowel\n",ch);
                break;
    }
    getch();
}
```

### **OUTPUT**

#### **Run 1:**

Enter any character E  
E is a vowel

#### **Run 2:**

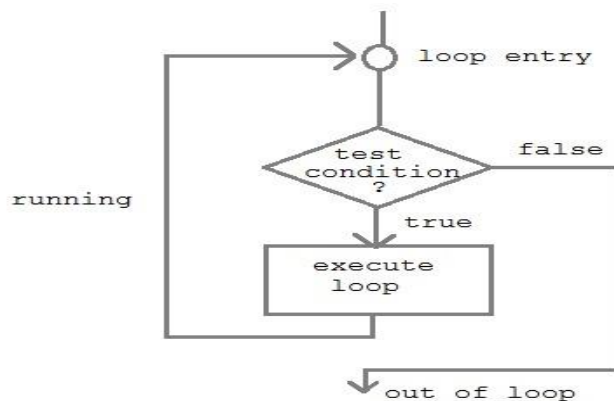
Enter any character s  
s is not a vowel

## LOOPING OR ITERATIVE OR REPETITIVE STATEMENTS

Consider a situation in which we execute a single statement or block of statements repeatedly for required number of times. Such kind of problems can be solved using **looping** statements in C.

For example, assume a situation where we print a message for 100 times. If we want to perform that task without using looping statements, we have to either write 100 printf statements or we have to write the same message for 100 times in a single printf statement. Both are complex methods. The same task can be performed very easily using looping statements.

The process of *repeatedly executing* a single statement or collection of statement until the given *condition is FALSE* is called looping. The statements gets executed many number of times based on the condition. But if the condition is given in such a logic that the repetition continues any number of times with no fixed condition to stop looping those statements, then this type of looping is called *infinite looping*.



### C language provides four looping statements

1. while statement
2. do-while statement
3. for statement
4. Nested for loop

### While or Entry Control or Pre Check Loop

The while loop is used to repeat a block of statements for given number of times, until the given condition is False. While loop start with the condition, if the condition is True then statements inside the while loop will be executed. If the given condition is false then it will not be executed at *least once*. It means, while loop may execute zero or more time and the

#### Syntax

```
while ( Condition )  
{  
    statement 1;  
    statement 2;  
    .....  
}
```

Where

**Statement:** simple or compound statement

**while:** keyword

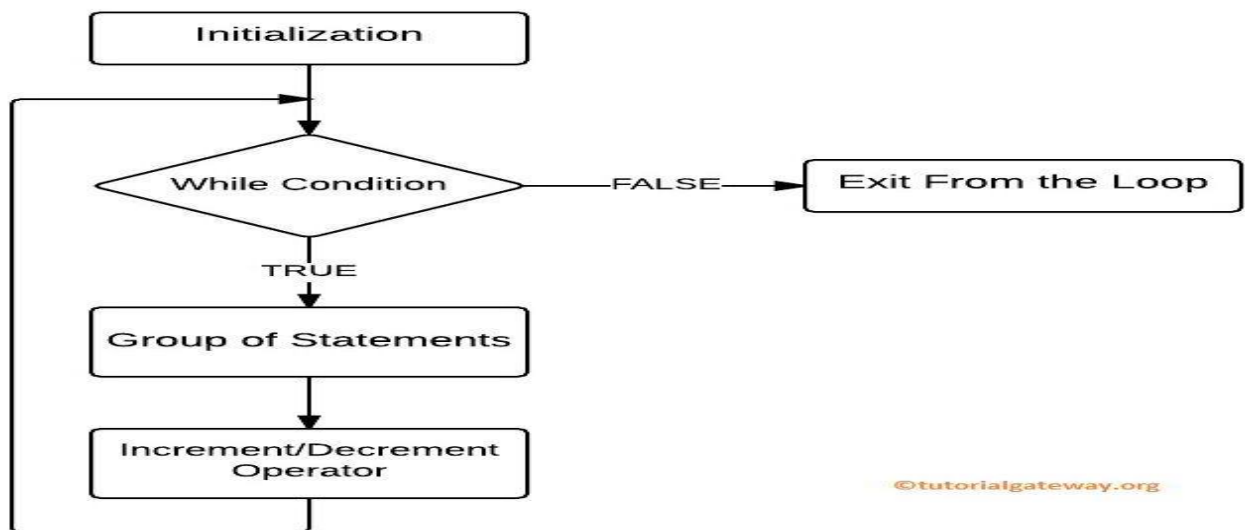
**Condition :** results in TRUE or FALSE

This is the statement Outside the While Loop but inside the main() Function

First compiler will check for the condition inside the While loop. If the Condition is True then the statement or group of statements under the while loop block will be executed. If the Condition is False then compiler will come out of the loop and execute other statements outside the while loop.

**TIP:** For single statement, curly braces are not required but if we omit them for multiple statements then it will execute the first statement only. It is always good practice to use braces all the time.

### Flow Chart for While loop in C Programming



- While loop will check for the condition at the beginning of the loop.
- If the condition is True then it will execute the statements inside the loop.
- Next we have to use Increment and Decrement Operator inside the while loop to increment and decrements the value. Please refer Increment and Decrement Operator in C article to understand the functionality
- Again it will check for the condition after the value incremented. As long as the condition is True, the statements inside the while loop will be executed.
- If the condition is False then it will exit from the While loop

**//Program to display 1 to n natural numbers and its sum-using while loop.**

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

```
    int n,i,sum=0;
```

```
clrscr();
printf("Enter the stop value for natural number series\n");
scanf("%d",&n);
i=1; //initialize
while(i<=n) //condition check
{
    printf("%d\t",i);
    sum=sum+i;
    i++; //increment i by 1
}
printf("\nSum of %d natural numbers=%d",n,sum);
getch();
}
```

### **OUTPUT**

Enter the stop value for natural number series

100

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sum of 100 natural numbers=5050

### **Points to be Remembered**

When we use while statement, we must follow the following...

- ✓ **while** is a keyword so it must be used only in lower case letters.
- ✓ If the condition contains variable, it must be assigned a value before it is used.
- ✓ The value of the variable used in condition must be modified according to the requirement inside the while block.
- ✓ In while statement, the condition may be a direct integer value, a variable or a condition.
- ✓ A while statement can be an empty statement.

### **Do While or Exit Control or Post check loop**

The Do While loop in C Programming will test the given condition at the end of loop so, Do While loop executes the statements inside *the code block at least once even if the given condition Fails*.

The While loop that test the condition before entering into the code block. If the condition is True then only statements inside the loop will be executed otherwise, statements will not be executed at least once. There are some situation

where it is necessary to perform some operations (execute some statements) first and then check for the condition. In these cases we can go for Do While loop.

**do while Syntax**

**The syntax of the Do While Loop in C Programming is as shown below:**

```
do
{
    statement 1;
    statement 2;
    .....
    statement n;
} while (condition);
```

Where

**Statement:** simple or compound statement

**do:** keyword

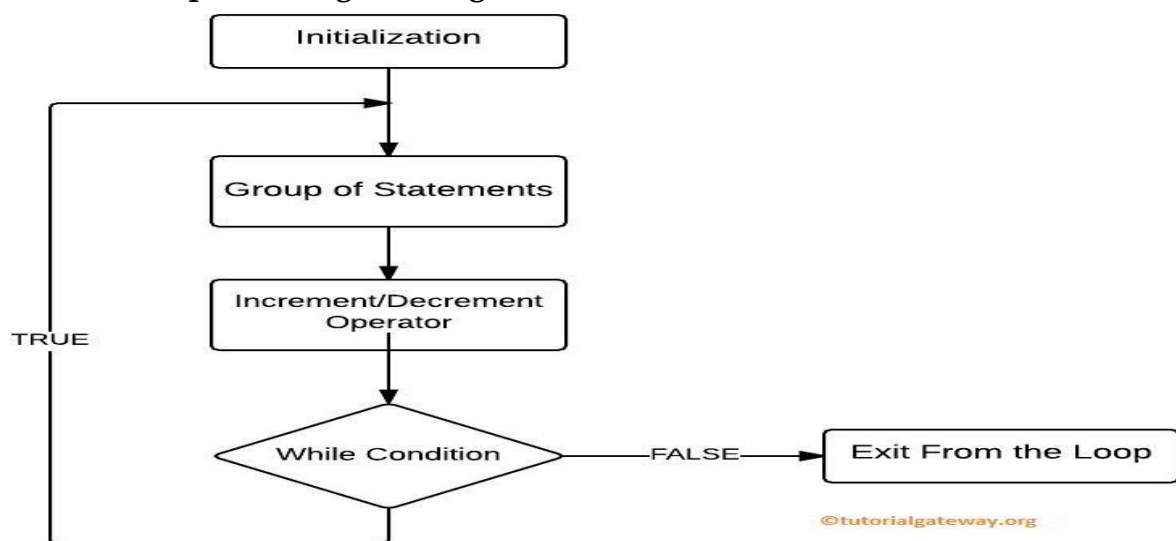
**while:** keyword

**Condition :** results in TRUE or FALSE

First it will execute the statements inside the loop and then after reaching the end, it will check the condition inside the while. If the condition is *true* then it will repeat the process. If the condition fails then Do While Loop will be terminated.

**NOTE:** We have to place semi-colon after the While condition.

**Do While Loop in C Programming Flow Chart**



**Flow chart sequence is:**

- First, we initialize our variables, next it will enter into the Do While loop.
- It will execute the group of statements inside the loop.
- Next, we have to use Increment and Decrement Operator inside the loop to increment or decrements the value. Please refer Increment and Decrement Operator in C article to understand increment and decrement operator

- Now it will check for the condition. If the condition is **True**, then the statements inside the ***do while loop*** will be executed again. It will continue the process as long as the condition is **True**.
- If the condition is **False** then it will exit from the loop.



नहि ज्ञानेन सद्रशं  
**SHREE MEDHA DEGREE COLLEGE**



**// Program to print the Fibonacci series for first n terms using do while loop**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a ,b,c,i,n;
    clrscr();
    printf("Enter the final value\n");
    scanf("%d",&n);
    a=0; b=1;
    printf(" Fibonacci Series....\n");
    if(n==0)
    printf("0\n");
    else if(n==1)
    printf("0\n1\n");
    else
    {
        printf("%d\n",a);
        printf("%d\n",b);
        i=2;
        do
        {
            c=a+b;
            printf("%d\n",c);
            a=b;
            b=c;
            i++;
        } while(i<=n);
    }
    getch();
}
```

### OUTPUT

#### **Run 1:**

Enter the final value 7

Fibonacci value.... 0 1 1 2 3 5 8

#### **Run 2:**

Enter the final value 1

Fibonacci value.... 0 1

### Points to be remembered

#### When we use do-while statement, we must follow the following...

- ✓ Both **do** and **while** are keywords so they must be used only in lower case letters.
- ✓ If the condition contains variable, it must be assigned a value before it is used.
- ✓ The value of the variable used in condition must be modified according to the requirement inside the do block.
- ✓ In do-while statement the condition may be, a direct integer value, a variable or a condition.
- ✓ A do-while statement can be an empty statement.
- ✓ **In do-while, the block of statements are executed atleast once.**

### For Loop in C Programming

The for loop is used to repeat a block of statements for *given number of times*, until the given condition is False. For loop is one of the mostly used loop in any programming language.

#### For loop Syntax

for (initialization; test condition; increment/decrement operator)

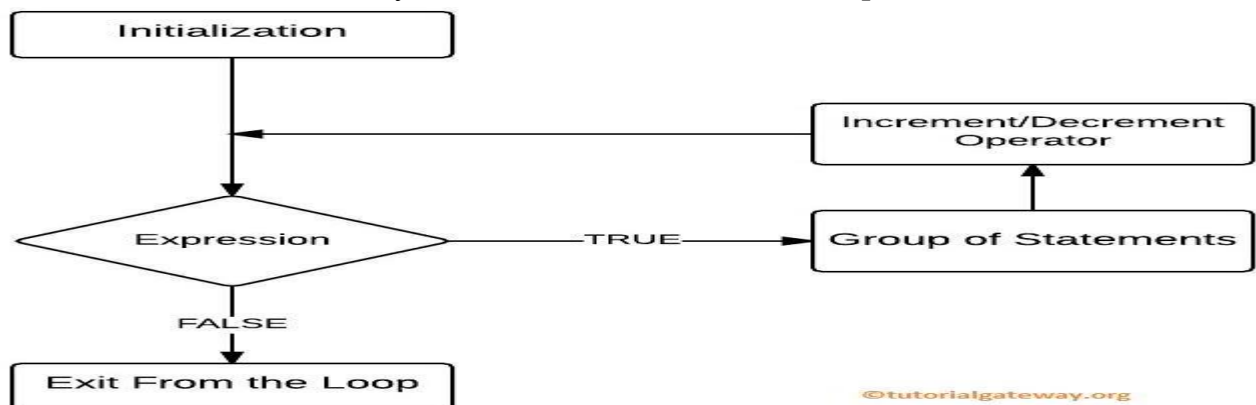
```
{  
    //Statement 1  
    //Statement 2  
    .....  
    //Statement n  
}
```

If you observe the above syntax, In for loop there are three expressions separated by the semi-colons (;) and the execution of these expressions are as follows:

- **Initialization:** For loop starts with the initialization statement so, initialization of counters variables is done first (For example counter = 1 or i = 1.). The initialization section is executed only once at the beginning of the for loop.
- **Test Condition:** The value of the counter variable is tested against the test condition. If the condition is True then it will execute the statements inside the For loop. If the condition fails, then for loop will be terminated.
- **Increment and decrement operator:** This expression is executed after the end of each iteration. This operator helps to increase or decrease the counter variable as per our requirement. Please refer Increment and decrement operator in C article to understand the operator

Flow Chart for loop in C

Below screenshot will show you the flow chart of the For Loop

The execution process of the for loop is:

**Initialization:** We initialize the counter variable(s) here. For example  $i=1$ .

**Test condition:** It will check for the condition against the counter variable. If the condition is True then it will execute the statements inside the for loop. If the condition is False then it will exit from the loop

After completing the iteration it will execute the Increment and Decrement Operator inside the for loop to increment or decrement the value.

Again it will check for the condition after the value incremented. As long as the condition is True, the statements inside the for loop will be executed.

//Program to find the factorial of a given number using for loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    long int fact=1,n,i;
    clrscr();
    printf("Enter the number\n");
    scanf("%ld",&n);
    if(n<=0)
        fact=1;
    else
    {
        for(i=1;i<=n;i++)
            fact=fact*i;
    }
    printf("\nFactorial of %ld=%ld\n",n,fact);
    getch();
}
  
```

OUTPUT

Enter the number 7  
Factorial of 7=5040

**POINTS TO BE REMEMBERED**

**When we use for statement, we must follow the following...**

- ✓ **for** is a keyword so it must be used only in lower case letters.
- ✓ Every for statement must be provided with initialization, condition and modification (They can be empty but must be separated with ";")  
Ex: **for ( ; ; )** or **for ( ; condition ; modification )** or **for ( ; condition ; )**
- ✓ In for statement, the condition may be a direct integer value, a variable or a condition.
- ✓ The for statement can be an empty statement.

**NESTED FOR LOOP**

A **for loop** inside another **for loop** is called a **nested for loop**. We can have any number of **nested loops** as required. Consider a **nested loop** where the outer **loop** runs  $n$  times and consists of another **loop** inside it. The inner **loop** runs  $m$  times. Then, the total number of times the inner **loop** runs during the program execution is  $n*m$ .

**Syntax of Nested for loop**

```
for (initialization; condition; increment/decrement)
{
    statement(s);
    for (initialization; condition; increment/decrement)
    {
        statement(s);
        ... ..
    }
    ... ..
}
```

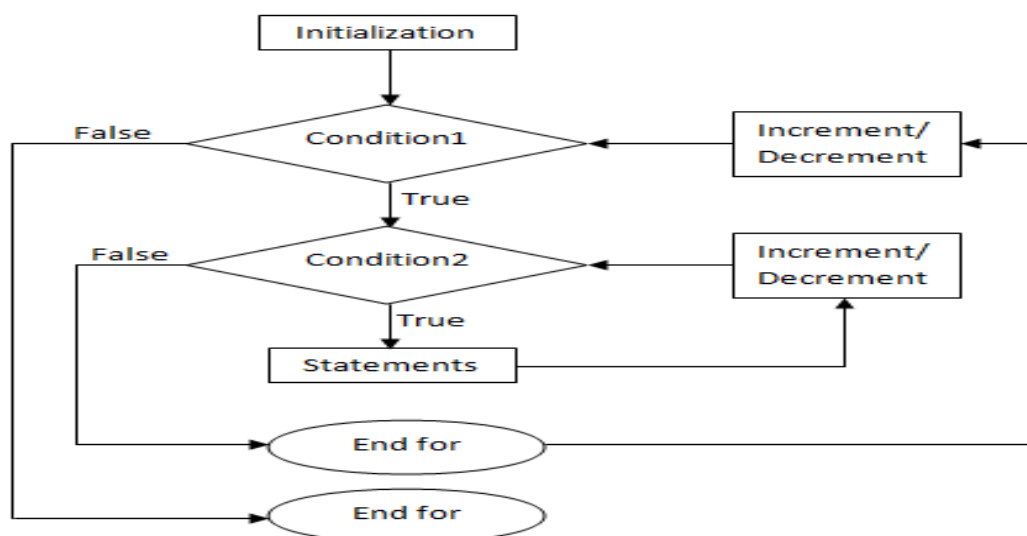
**Flowchart of Nested for loop**

Fig: Flowchart for nested for loop

We have nested a for loop inside another for loop, this is called nesting of loops. Such type of nesting is often used for handling multidimensional arrays.

**//Program to illustrate nested for loop.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    clrscr();
    for(i=0;i<5;i++)
    {
        for(j=0;j<=i;j++)
        {
            printf("*");
        }
        printf("\n");
    }
    getch();
}
```

**OUTPUT**

```
*
**
***
****
*****
```

### ***Difference between while loop and do while loop***

Sl.no	While	Do while
1	In While loop the condition is tested first and then the statements are executed if the condition turns out to be true.	In do while the statements are executed for the first time and then the conditions are tested, if the condition turns out to be true then the statements are executed again.
2	while is more commonly used.	A do while is used for a block of code that must be executed at least once. These situations tend to be relatively rare.
3	while loop do not run in case the condition given is false	A do while loop runs at least once even though the the condition given is false
4	In a while loop the condition is first tested and if it returns true then it goes in the loop	In a do-while loop the condition is tested at the last.
5	While loop is entry control loop	do while is exit control loop.

<b>6</b>	It is called as Pre Check loop	It is called as Post Check loop
<b>7</b>	while (condition) { Statements; }	do { Statements; }while(condition);

### **Un-conditional control statements or Jumping or Skipping**

In c, there are control statements which does not need any condition to control the program execution flow. These control statements are called as **unconditional control statements**. C has four **statements** that perform an **unconditional** control transfer. They are *return, goto, break and continue*. Of these, return is used only in functions. The goto and return may be used anywhere in the program but continue and break **statements** may be used only in conjunction with a loop **statement**.

*C programming language provides the following unconditional control statements...*

- break
- continue
- goto
- return

### **break statements**

In C programming, break statement is used with conditional if statement. The break is used in terminating the loop immediately after it is encountered. it is also used in switch...case statement. which is explained in next topic.

1. break statement is used to terminate switch case statement
2. break statement is also used to terminate looping statements like while, do-while and for.

```

while ( condition)
{
    ...
    break ;
    ...
}

do
{
    ...
    break ;
    ...
} while ( condition) ;
    
```

```

for (initilization; condition; modification)
{
    ...
    break ;
    ...
}
    
```

**//Program to implement break statement**

```

#include <stdio.h>
#include<conio.h>
void main(){
    char ch ;
    clrscr() ;
    do
    {
        printf("Enter Y / N : ") ;
    }
    while (ch != 'N') ;
}
    
```



```

scanf("%c", &ch) ;
if(ch == 'Y')
{
    printf("Okay!!! Repeat again !!!\n") ;
}
else
{
    printf("Okay !!! Breaking the loop !!!") ;
    break ;
}
} while( 1 ) ;
getch() ;
}

```

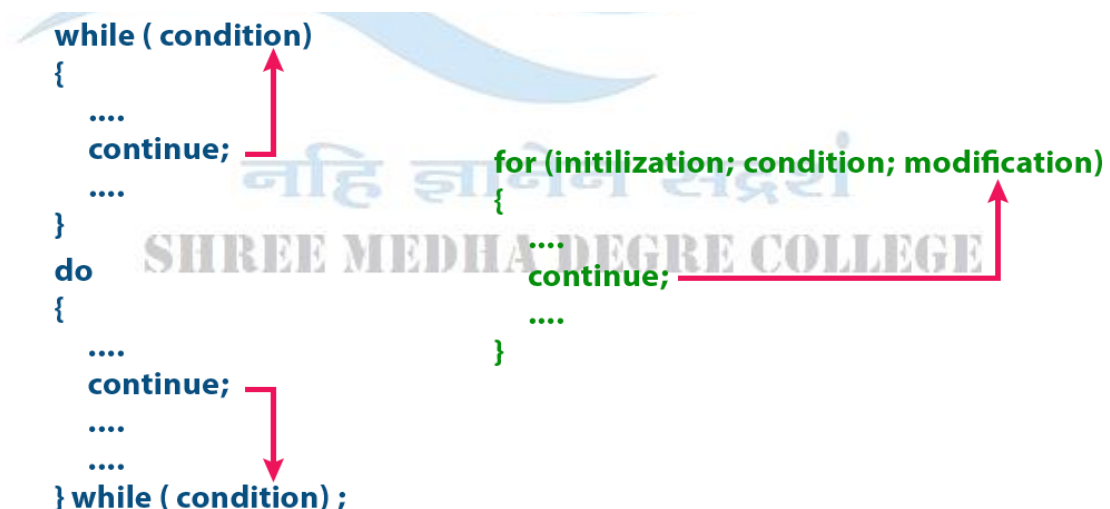
**OUTPUT**

Enter Y / N : Y  
 Okay!!! Repeat again !!!  
 Enter Y / N : Y  
 Okay!!! Repeat again !!!

**continue statement**

The **continue** statement is used to move the program execution control to the beginning of looping statement. When **continue** statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The **continue** statement can be used with looping statements like while, do-while and for.

When we use **continue** statement with **while** and **do-while** statements the execution control directly jumps to the condition. When we use **continue** statement with **for** statement the execution control directly jumps to the modification portion (increment / decrement / any modification) of the for loop. The **continue** statement execution is as shown in the following figure.



```
//Program to illustrate continue statement.
#include <stdio.h>
#include <conio.h>
void main(){
    int number ;
    clrscr() ;
    while( 1 )
    {
        printf("Enter any integer number: ") ;
        scanf("%d", &number) ;
        if(number%2 == 0)
        {
            printf("Entered number is EVEN!!! Try another number!!!\n") ;
            continue ;
        }
        else
        {
            printf("You have entered ODD number!!! Bye!!!") ;
            exit(0) ;
        }
    }
    getch() ;
}
```

### OUTPUT

Enter any integer numbers: 50  
Entered number is EVEN!!! Try another number!!!  
Enter any integer number: 15  
You have entered ODD number!!! Bye!!!

### *goto statements***statement**

The **goto** statement is used to jump from one line to another line in the program. Using **goto** statement we can jump from top to bottom or bottom to top. To jump from one line to another line, the goto statement requires a **label**. Label is a name given to the instruction or line in the program. When we use **goto** statement in the program, the execution control directly jumps to the line with specified label.

#### *Syntax:*

goto **label**;

.....  
.....  
.....

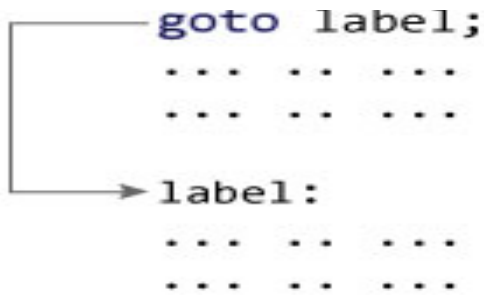
#### **label:**

statement;

*In this syntax, label is an identifier.*

When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code below it.

## Forward



## Backward



//Program to illustrate goto statement.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int age;
    Vote:
    printf("you are eligible for voting");
    NoVote:
    printf("you are not eligible to vote");
    printf("Enter you age:");
    scanf("%d", &age);
    if(age>=18)
        goto Vote;
    else
        goto NoVote;
    getch();
}
```

**Explanation:** In the above example, Vote and NoVote are labels. When the input is  $\geq 18$ , the goto statement is transferring the control to label - Vote, otherwise it transfers the control to label-NoVote.

### Points to be remembered

*When we use break, continue and goto statements, we must follow the following...*

- The **break** is a keyword so it must be used only in lower case letters.
- The **break** statement cannot be used with **if** statement.
- The **break** statement can be used only in switch case and looping statements.
- The **break** statement can be used with **if** statement, only if that **if statement** is written inside the switch case or looping statements.
- The **continue** is a keyword so it must be used only in lower case letters.
- The **continue** statement is used only within **looping statements**.
- The **continue** statement can be used with **if** statement, only if that **if statement** is written inside the looping statements.
- The **goto** is a keyword so it must be used only in lower case letters.
- The **goto** statement must require a **label**.
- The **goto** statement can be used with any statement like if, switch, while, do-while and for etc.,

## **Array**

Array is a collection of variables belonging to the same data type. You can store group of data of same data type in an array. Array is set of homogenous elements.

- **It allocates sequential memory locations.**
- **Individual values are called as elements.**

## **Need of array**

Consider a scenario wherein you have to store 100 integer numbers, entered by user, in order to find out the average of them. To program this scenario, you have two ways – 1) Define 100 variable of integer type and at last perform the average operation. 2) Have a single integer array to store all the values.

Which solution is better as per you? Obviously the second solution, it is convenient to store same data types in one single variable and later access them using array index.

- Array might be belonging to any of the data types
- Array size must be a constant value.
- Always, Contiguous (adjacent) memory locations are used to store array elements in memory.
- It is a best practice to initialize an array to zero or null while declaring, if we don't assign any values to array.

## **Some examples where arrays can be used are**

- 1) List of temperatures recorded every hour in a day, or a month, or a year.
- 2) List of employees in a company.
- 3) List of products and their cost sold by a store.
- 4) List of students in a class.
- 5) List of customers and their telephone numbers.

## **Characteristics of Arrays in C**

- 1) An array holds elements that have the same data type.
- 2) Array elements are stored in subsequent memory locations.
- 3) Two-dimensional array elements are stored row by row in subsequent memory locations.
- 4) Array name represents the address of the starting element.
- 5) Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.

## **TYPES OF C ARRAYS:**

There are 3 types of C arrays. They are,

1. One dimensional array.
2. Two dimensional array.
3. Multi-dimensional array
  - a. Three dimensional array
  - b. four dimensional array etc...

## 1. ONE DIMENSIONAL or 1-D ARRAY

- Single or One Dimensional array is used to represent and store data in a linear form.
- Array having only one subscript variable is called **One-Dimensional array**
- It is also called as **Single Dimensional Array** or **Linear Array**

**Syntax:**

<b>datatype array_name[array_size];</b>
---

- **datatype:** What kind of values array can store (Example: int, float, char).
- **array\_name:** To identify the array.
- **array\_size(INDEX):** The maximum number of values that the array

can hold.

**Example:**

```
int number[5];  
number
```



**MEMORY REPRESENTATION OF ONE DIMENSIONAL ARRAY**

- The above statement declares a number variable to be an array.  
containing 5 elements.

- The array **INDEX** (also known as **subscripts**) starts from **zero**.
- This means that the array **number** will contain 5 elements in all.
- The first element will be stored in `number[0]`.
- The second element will be stored in `number[4]`.

### **Initialization of one dimensional array**

- Arrays may be initialized when they are declared, just as any other variables.
- Place the initialization data in curly { } braces following the equals sign. Note the use of commas in the examples below.
- An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.
- If an array is to be completely initialized, the dimension of the array is not required. The compiler will automatically size the array to fit the initialized data.

**Syntax for creating an array with size and initial values**

<b>datatype arrayName [ size ] = {value1, value2, ...} ;</b>
--

**Syntax for creating an array without size and with initial values**

<b>datatype arrayName [ ] = {value1, value2, ...} ;</b>
---

In the above syntax, the **datatype** specifies the type of values we store in that array and **size** specifies the maximum number of values that can be stored in that array. The values in the list are separated by commas.

**Example**

```
int number [5] = { 35, 20, 40, 57, 19 };
```

Will declare the array size as an array of size 5 and will assign different values to each element if the number of values in the list is less than the number of elements, then only that many elements are initialized. The remaining elements will be set to zero automatically.

Storing Values after Initialization	
35	Number[0]
20	Number[1]
40	Number[2]
57	Number[3]
19	Number[4]

In the declaration of an array the size may be omitted, in such cases the compiler allocates enough space for all initialized elements.

**Example**

```
int counter [ ] = {1,1,1,1};
```

Will declare the array to contain four elements with initial values 1. this approach works fine as long as we initialize every element in the array.

**Example**

```
char studentName [ ] = "MedhaBCA";
```

In the above example declaration, size of the array **studentName** is 8. This is because in case of character array, compiler stores one extra character called **\0** (NULL) at the end.

*The initialization of arrays in c suffers two drawbacks*

1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method to initialize large number of elements.

```
// Program to accept 5 array elements and display it.
```

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

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

```
void main()
```

```
{
```

```
    int a[5],i;
```

```
    clrscr();
```

```
    printf("Enter the array elements\n");
```

```
    for(i=0;i<5;i++)
```

```
        scanf("%d",&a[i]);
```



```

printf("Array elements are...\n");
for(i=0;i<5;i++)
printf("a[%d]=%d\n",i,a[i]);
getch();
}

```

## -----OUTPUT-----

Enter the array elements 10 20 30 40 50

Array elements are....

a[0]=10

a[1]=20

a[2]=30

a[3]=40

a[4]=50

## 2. TWO DIMENSIONAL or 2-D ARRAY

- Two Dimensional array is used to represent and store data in a matrix form.
- Array having two subscripts variable is called **Two-Dimensional array**
- It is also called as **Multi-Dimensional Array**
- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns.

**Syntax:**

```
datatype array_name[row_size] [column_size];
```

- **datatype:** What kind of values array can store (Example: int, float, char).
- **array\_name:** To identify the array.
- **row\_size(INDEX):** The maximum number of rows that the array can hold.
- **column\_size(INDEX):** The maximum number of columns that the array can hold.

**Example:**

```
int number[3][3];
```

- The above statement declares a number variable to be an array containing 5 elements.
- The array **INDEX** (also known as **subscripts**) starts from **zero**.
- This means that the array **number** will contain 5 elements in all.
- The first element will be stored in number[0].
- The second element will be stored in number[4].

**Reserved Space**

number	[0]	[1]	[2]
[0]			
[1]			
[2]			

**MEMORY REPRESENTATION OF TWO DIMENSIONAL ARRAY**

**Initialization of Two dimensional array**

We have divided the concept into three different types –

**Method 1: Initializing all Elements row wise**

For initializing 2D Array we can need to assign values to each element of an array using the below syntax.

```
datatype arrayName [rows][colmnns] = {
    {r1c1value, r1c2value, ...},
    {r2c1, r2c2,...},
    {rncn, rncn,...}
};
```

**Example:**

```
int number[3][3] = {
    { 1 , 4 , 8 },
    { 5 , 2 , 9 },
    { 6 , 5 , 10 }
};
```

The following array: Storing values after initialization

number	[0]	[1]	[2]
[0]	1	4	8
[1]	5	2	9
[2]	6	5	10

**Method 2: Combine and Initializing 2D Array**

Initialize all Array elements but initialization is much straight forward. All values are assigned sequentially and row-wise

```
datatype arrayName [rows][columns] = {value1, value2, value3,.....,valuen} ;
```

**Example:**

```
int a[3][2] = {1 , 4 , 5 , 2 , 6 , 5};
```

**//Program to read a matrix and display its transpose implementation of 2D array.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10][10],r,c,i,j;
    printf("\n Enter the order of the matrix");
    scanf("%d %d",&r,&c);
    printf("Enter %d matrix elements\n",r*c);
    for(i=0;i<r;i++)
    for(j=0;j<c;j++)
    scanf("%d",&a[i][j]);
    printf("Matrix A is\n");
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            printf("%3d",a[i][j]);
        }
        printf("\n");
    }
    printf("Transpose of A is\n");
    for(i=0;i<c;i++)
    {
        for(j=0;j<r;j++)
        {
            printf("%3d",a[j][i]);
        }
        printf("\n");
    }
    getch();
}
```

-----OUTPUT-----

Enter the order of the matrix 2 3

Enter 6 matrix elements

1 2 3

4 5 6

Matrix A is

1 2 3

4 5 6

Transpose of A is

1 4

2 5

3 6

## Applications of Arrays

### **Arrays are used to Store List of values**

In c programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.

### **Arrays are used to Perform Matrix Operations**

We use two dimensional arrays to create matrix. We can perform various operations on matrices using two dimensional arrays.

### **Arrays are used to implement Search Algorithms**

We use single dimensional arrays to implement search algorithms like ...

- Linear Search
- Binary Search

### **Arrays are used to implement Sorting Algorithms**

We use single dimensional arrays to implement sorting algorithms like

...

- Insertion Sort
- Bubble Sort
- Selection Sort
- Quick Sort
- Merge Sort, etc.,

### **Arrays are used to implement Data structures**

We use single dimensional arrays to implement data structures like...

- Stack Using Arrays
- Queue Using Arrays

### **Arrays are also used to implement CPU Scheduling Algorithms**

## STRING

- C Strings are nothing but array or collection of characters ended with null character ('\0').
- This null character indicates the end of the string.
- Strings are always enclosed by double quotes. Whereas, character is enclosed by single quotes in C.

EXAMPLE FOR C STRING:

SHREE MEDHA DEGREE COLLEGE

### **Memory Representation of Above Defined String in C**

c	l	o	u	d	\0
---	---	---	---	---	----

### **Declaration of strings:**

Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string.

```
char str_name[size];
```

In the above syntax `str_name` is any name given to the string variable and `size` is used to define the length of the string, i.e. the number of characters' strings will store.

Please keep in mind that there is an extra *terminating character* which is the *Null character* (`'\0'`) used to indicate termination of string which differs strings from normal character arrays.

**Example: `char str [50],name [25];`**

### **Initializing String [Character Array]:**

Whenever we declare a String then it will contain garbage values inside it. We have to initialize String or Character array before using it. Process of Assigning some legal default data to String is Called Initialization of String. There are different ways of initializing String in C Programming –

1. Initializing Unsized Array of Character
2. Initializing String Directly
3. Initializing String Using Character Pointer

#### **Way 1: Unsized Array and Character**

- Unsized Array: Array Length is not specified while initializing character array using this approach
- Array length is Automatically calculated by Compiler
- Individual Characters are written inside Single Quotes, separated by comma to form a list of characters. Complete list is wrapped inside Pair of Curly braces
- Please Note: NULL Character should be written in the list because it is ending or terminating character in the String/Character Array

**`char name [] = {'M','E','D','H','A','\0'};`**

#### **Way 2: Directly initialize String Variable**

- In this method we are directly assigning String to variable by writing text in double quotes.
- In this type of initialization, we don't need to put NULL or Ending / Terminating character at the end of string. It is appended automatically by the compiler.

**`char name [] = "MEDHA";`**

#### **Way 3: Character Pointer Variable**

- 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 `name [0]` )
- NULL Character is appended Automatically

**`char *name = "MEDHA";`**

### **gets() function : Reading or Accepting String From User in C**

Reads characters from the standard input (stdin) and stores them as a C string into `str` until a newline character or the end-of-file is reached.



**Syntax for Accepting String:**

```
char * gets ( char * str );
```

OR

```
gets( <variable-name> )
```

**Live Example:**

```
#include<stdio.h>
void main ( )
{
char name [20];
printf("\n Enter the Name : ");
gets(name);
}
```

**Explanation:**

- Whenever gets() statement encounters then characters entered by user (the string with spaces) will be copied into the variable.
- If user start accepting characters, and if new line character appears then the newline character will not be copied into the string variable (i.e name).
- A terminating null character is automatically appended after the characters copied to string variable (i.e name)
- 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 (which can lead to buffer overflows).

**Some Legal Declarations:**

Statement	Use	Size	Terminates
gets(name)	Accepting Name of Person with Spaces	No Specification	After Enter Key
gets(city)	Accepting Name of City	No Specification	After Enter Key
gets(address)	Accepting address with Spaces	No Specification	After Enter Key
gets(bname)	Accepting book name	No Specification	After Enter Key

**Some Rules and Facts:****A. %s is not Required:**

Like scanf statement %s is not necessary while accepting string.

```
scanf("%s",name);
```

and here is gets() syntax which is simpler than scanf() -

```
gets(name);
```

**B. Spaces are allowed in gets() :**

```
gets(name);
```

Whenever the above line encounters then interrupt will wait for user to enter some text on the screen. When user starts typing the characters then all characters will be copied to string and when user enters newline character then process of accepting string will be stopped.

**Sample Input Accepted by Above Statement:**

Value Accepted: MEDHA COLLEGE\n

Value Stored: MEDHA COLLEGE (\n Neglected)

**Scanf : Reading or Accepting String From User in C**

In C Programming we can use scanf function from stdio.h header file to read string. Scanf function is commonly used for accepting string.

**Syntax for Accepting String:**

```
scanf("%s",Name_Of_String_Variable);
```

**Live Example:**

```
#include<stdio.h>
void main()
{
char name[20];
printf("\nEnter the Name : ");
scanf("%s",name);
}
```

**Explanation: Scanf Program**

In the above program we are accepting string from user using scanf() function. We cannot use string ampersand address operator in the scanf() because string itself can be referred by address.

**printf >> Displaying String in C Programming:****Syntax:****Way 1: Messaging**

```
printf (" Type your Message / Instruction ") ;
```

**Way 2: Display String**

```
printf ("Name of Person is %s ", name ) ;
```

**Notes or Facts:**

1. printf is included in header file "stdio.h"
2. As name suggest it used for Printing or Displaying Messages or Instructions

**Uses:**

1. Printing Message
2. Ask user for entering the data (Labels. Instructions)
3. Printing Results



**Live Example:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10];
printf("Enter the String : ");

scanf("%s",str); // Accept String
printf("String is : %s ",str);
getch();
}
```

**Output:**

```
Enter the String : Pritesh
String is : Pritesh
```

**Puts >> Displaying String in C Programming :**

**Way 1 :Messaging**

```
puts (" Type your Message / Instruction ");
```

- Like printf Statement puts() can be used to display message.

**Way 2: Display String**

```
puts(string_Variable_name) ;
```

**Notes or Facts:**

- puts is included in header file “stdio.h”
- As name suggest it used for Printing or Displaying Messages or Instructions

**Uses:**

- Printing Message
- Ask user for entering the data (Labels. Instructions)
- Printing Results

**Live Example:**

```
#include< stdio.h>
#include< conio.h>
void main()
{
char string [ ] = "This is an example string\n";
puts(string); // String is variable Here
puts("String"); // String is in Double Quotes
getch();
}
```

**Output:**

```
String is: This is an example string
String is: String
```

### Arithmetic Operations On Character:

- C Programming Allows you to Manipulate on String
- Whenever the Character is variable is used in the expression then it is automatically Converted into Integer Value called ASCII value
- All Characters can be Manipulated with that Integer Value. (Addition, Subtraction)

#### Examples:

- ASCII value of: 'a' is 97
- ASCII value of: 'z' is 121

#### Possible Ways of Manipulation:

##### Way 1: Displays ASCII value [ Note that %d in printf ]

```
char x = 'a';  
printf("%d",x); // Display Result = 97
```

##### Way 2: Displays Character value [ Note that %c in printf ]

```
char x = 'a';  
printf("%c",x); // Display Result = a
```

##### Way 3: Displays Next ASCII value [ Note that %d in printf ]

```
char x = 'a' + 1 ;  
printf("%d",x);  
// Display Result = 98 ( ascii of 'b' )
```

##### Way 4: Displays Next Character value [Note that %c in printf ]

```
char x = 'a' + 1;  
printf("%c",x); // Display Result = 'b'
```

##### Way 5: Displays Difference between 2 ASCII in Integer [Note %d in printf ]

```
char x = 'z' - 'a';  
printf("%d",x);  
/* Display Result = 25  
(difference between ASCII of z and a) */
```

##### Way 6: Displays Difference between 2 ASCII in Char [Note that %c in printf ]

```
char x = 'z' - 'a';  
printf("%c",x);  
/* Display Result = ↓  
(difference between ASCII of z and a) */
```

**NULL Character: Terminating Character in String**

We have already studied how string is declared and initialized now in this chapter we will be learning about terminating character of String i.e NULL Character.

**Why do we need Terminating Character?**

- C Programming does not provide inbuilt '**String**' data type. String is one of the most important and necessary **Data Structure** in C Programming
- We can say String as **Variable Length** Structure stored in **Fixed Length** Structure
- Array Size is not the Actual Length of the String, so to recognize the End of the String we use **NULL character**

**Pictorial Understanding:****Explanation:**

In the above picture we can see –

- Before Initializing String, it Contain **Garbage Characters**.
- Length of String [i.e Declared character array] is 7 so it Contain 7 Garbage Characters.
- When we attempt to store "CAT" in the string of Character length 7 then String Would be "CAT@'af"

Location	Value Before Initialization	Value After Initialization
name[0]	a	C
name[1]	;	A
name[2]	#	T
name[3]	`	NULL Character
name[4]	@	@
name[5]	a	a
name[6]	f	f

Since valid String is "CAT" whenever we are trying to access the array using character array variable then valid string before NULL character will be considered as Valid.

**Packing Up: Summary**

Without NULL character string stored will be	CAT'@af
NULL Character	'\0'
ASCII Value of NULL Character	0

## **STRING HANDLING/MANIPULATION FUNCTIONS**

**string.h** header file supports all the string functions in C language. All the string functions are given below.

### **C - strlen() function**

- strlen() function in C gives the length of the given string. Syntax for strlen() function is given below.  
len = strlen ( str );
- strlen() function counts the number of characters in a given string and returns the integer value.
- It stops counting the character when null character is found. Because, null character indicates the end of the string in C.

### **C - strlwr() function**

- strlwr() function converts a given string into lowercase. Syntax for strlwr() function is given below.  
char strlwr(string);
- strlwr() function is non standard function which may not available in standard library in C.

### **C -strupr() function**

- strupr() function converts a given string into uppercase. Syntax forstrupr() function is given below.  
char \*strupr(char \*string);
- strupr() function is non standard function which may not available in standard library in C.

### **C - strrev() function**

- strrev() function reverses a given string in C language. Syntax forstrrev() function is given below.  
char \*strrev(char \*string);
- strrev() function is non standard function which may not available in standard library in C.

### **C - strdup() function**

- strdup() function in C duplicates the given string. Syntax forstrdup() function is given below.  
char \*strdup(const char \*string);
- strdup() function is non standard function which may not available in standard library in C.

### **C - strcmp() function**

- strcmp() function in C compares two given strings and returns zero if they are same.

- If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value. Syntax for strcmp( ) function is given below.  
int strcmp ( const char \* str1, const char \* str2 );
- strcmp( ) function is case sensitive. i.e, "A" and "a" are treated as different characters.

**Return Type**

Return Type	Condition
-ve Value	String1 < String2
+ve Value	String1 > String2
0 Value	String1 = String2

**C – strcmpi() function**

- strcmpi( ) function in C is same as strcmp() function. But, strcmpi( ) function is not case sensitive. i.e, "A" and "a" are treated as same characters. Where as, strcmp() function treats "A" and "a" as different characters.
- strcmpi() function is non standard function which may not available in standard library in C.
- Both functions compare two given strings and returns zero if they are same.
- If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value. Syntax for strcmp( ) function is given below.  
int strcmpi ( const char \* str1, const char \* str2 );

**C – strcat() function**

- strcat( ) function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for strcat( ) function is given below.  
char \* strcat ( char \* destination, const char \* source );
- Example:  
strcat ( str2, str1 ); – str1 is concatenated at the end of str2.  
strcat ( str1, str2 ); – str2 is concatenated at the end of str1.
- As you know, each string in C is ended up with null character ('\0').
- In 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.

**C – strcpy() function**

- strcpy( ) function copies contents of one string into another string. Syntax for strcpy function is given below.  
char \* strcpy ( char \* destination, const char \* source );

- Example:  
strcpy ( str1, str2) – It copies contents of str2 into str1.  
strcpy ( str2, str1) – It copies contents of str1 into str2.
- If destination string length is less than source string, entire source string value won't be copied into destination string.
- For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

### **C – strncpy() function**

- strncpy( ) function copies portion of contents of one string into another string. Syntax for strncpy( ) function is given below.  
char \* strncpy ( char \* destination, const char \* source, size\_t num );
- Example:  
strncpy ( str1, str2, 4) – It copies first 4 characters of str2 into str1.  
strncpy ( str2, str1, 4) – It copies first 4 characters of str1 into str2.
- If destination string length is less than source string, entire source string value won't be copied into destination string.
- For example, consider destination string length is 20 and source string length is 30.
- If you want to copy 25 characters from source string using strncpy( ) function, only 20 characters from source string will be copied into destination string and remaining 5 characters won't be copied and will be truncated.

### **C – strncat() function**

- strncat( ) function in C language concatenates (appends) portion of one string at the end of another string. Syntax for strncat( ) function is given below.  
char \* strncat ( char \* destination, const char \* source, size\_t num );
- Example :  
strncat ( str2, str1, 3 ); – First 3 characters of str1 is concatenated at the end of str2.  
strncat ( str1, str2, 3 ); – First 3 characters of str2 is concatenated at the end of str1.
- As you know, each string in C is ended up with null character ('\0').
- In strncat( ) 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 strncat( ) operation.



### Introduction to Functions or Sub Routines

When we write a program to solve a larger problem, we divide that larger problem into smaller sub problems and are solved individually to make the program easier. In C, this concept is implemented using *functions*. Functions are used to divide a larger program into smaller *subprograms* such that program becomes easy to understand and easy to implement. A function is defined as follows...

*Function is a subpart of program used to perform specific task and it is executed individually.*

Every C program must contain at least one function called **main ( )**. However, a program may also contain other functions.

### Advantages of functions:

1. **Program development made easy:** Work can be divided among project members thus implementation can be completed in parallel.
2. **Program testing becomes easy :** Easy to locate and isolate a faulty function for further investigation
3. **Code sharing becomes possible:** A function may be used later by many other programs this means that a c programmer can use function written by others, instead of starting over from scratch.
4. **Code re-usability increases:** A function can be used to keep away from rewriting the same block of codes which we are going use two or more locations in a program. This is especially useful if the code involved is long or complicated.
5. **Increases program readability:** It makes possible top down modular programming. In this style of programming, the high-level logic of the overall problem is solved first while the details of each lower level functions is addressed later. The length of the source program can be reduced by using functions at appropriate places.
6. **Function facilitates procedural abstraction:** Once a function is written, it serves as a black box. All that a programmer would have to know to invoke a function would be to know its name, and the parameters that it expects
7. **Functions facilitate the factoring of code:** Every C program consists of one main( ) function typically invoking other functions, each having a well-defined functionality.

### Every function in C has the following

1. Function Declaration (Function Prototype)
2. Function Definition
3. Function Call



## Function Declaration or Function Prototype

The function declaration tells the compiler about function name, datatype of the return value and parameters. The function declaration is also called as *function prototype*. The function declaration is performed before main function or inside main function or inside any other function.

### Function declaration syntax -

<b>return_type function_name(parametersList or ArgumentList);</b>
---

In the above syntax, **return\_type** specifies the datatype of the value, which is sent as a return value from the function definition. The **function\_name** is a user-defined name used to identify the function uniquely in the program. The **parametersList** is the data values that are sent to the function definition.

## Function Definition

The function definition provides the actual code of that function. The function definition is also known as **body of the function**. The actual task of the function is implemented in the function definition. That means the actual instructions to be performed by a function are written in function definition. The actual instructions of a function are written inside the braces "{ }". The function definition is can be written before main function or after main function.

### Function definition syntax -

```
return_type function_name(parametersList or argumentList)
{
```

Actual code...

```
}
```

**Note:** *If the function definition is defined before main function then function prototyping is ignored or not required.*

## Function Call

The function call tells the compiler when to execute the function definition. When a function call is executed, the execution control jumps to the function definition where the actual code is executed and returns to the same functions call once the execution completes. The function call is performed inside main function, inside any other function, or inside the function itself.

### Function call syntax -

<b>functionName(parameters); of Functions</b>
---

- ✓ Using functions, we can implement modular programming.
- ✓ Functions makes the program more readable and understandable.
- ✓ Using functions, the program implementation becomes easy.
- ✓ Once a function is created, it can be used many times (**code re-usability**).
- ✓ Using functions larger program can be divided into smaller modules.

## Types of Functions in C

In C Programming Language, based on providing the function definition, functions are divided into two types. Those are as follows.

- System Defined Functions or Pre Defined.
- User Defined Functions

### **System Defined Functions**

The C Programming Language provides pre-defined functions to make programming easy. These *pre-defined* functions are also known as *system defined* functions. The system defined function is defined as follows...

*The function whose definition is defined by the system is called as system defined function.*

The system-defined functions are also called as *Library Functions or Standard Functions or Pre-Defined Functions*. The implementation of system-defined functions is already defined by the system.

In C, all the system defined functions are defined inside the header files like `stdio.h`, `conio.h`, `math.h`, `string.h` etc., For example, the functions `printf()` and `scanf()` are defined in the header file called `stdio.h`.

Whenever we use system-defined functions in the program, we must include the respective header file using `#include` statement. For example, if we use a system defined function `sqrt ( )` in the program, we must include the header file called *math.h* because the function `sqrt ( )` is defined in *math.h*.

### **Points to be remembered**

- ✓ System defined functions are declared in header files
- ✓ System defined functions are implemented in .dll files. (DLL stands for Dynamic Link Library).
- ✓ To use system-defined functions the respective header file must be included.

### **User Defined Functions**

In C programming language, users can also create their own functions. The functions that are created by users are called as user-defined functions. The user defined function is defined as follows...

*"The function whose definition is defined by the user is called as user defined function."*

That means the function that is implemented by user is called as user defined function. For example, the function `main` is implemented by user so it is called as user defined function.

In C every user defined function must be declared and implemented. Whenever we make function call the function definition gets executed. For example, consider the following program in which we create a function called `addition` with two parameters and a return value.

```
//program to implement functions
#include <stdio.h>
#include <conio.h>
int addition(int,int) ;// function declaration
void main()
{
    int num1, num2, result ;
    clrscr() ;
    printf("Enter any two integer numbers : ") ;
    scanf("%d%d", &num1, &num2);
    result = addition(num1, num2) ;// function call
    printf("SUM = %d", result);
    getch() ;
}

int addition(int a, int b) // function definition
{
    return a+b ;
}
```

In the above example program, the function declaration statement "int addition (int, int)" tells the compiler that there is a function with name addition, which takes two integer values as parameters and returns an integer value. The function call statement takes the execution control to the addition ( ) definition along with values of num1 and num2. Then function definition executes the code written inside it and comes back to the function call along with return value.

In the concept of functions, the function call is known as "Calling Function" and the function definition is known as "Called Function".

When we make a function call, the execution control jumps from calling function to called function. After executing the called function, the execution control comes back to calling function from called function. When the control jumps from calling function to called function it may carry one or more data values called "*Parameters*" and while coming back it may carry a single value called "*return value*". That means the data values transferred from calling function to called function are called as *Parameters* and the data value transferred from called function to calling function is called *Return value*.

Based on the data flow between the calling function and called function, the functions are classified as follows

### **Types of User Defined Functions**

- Function without Parameters and without Return value
- Function with Parameters and without Return value
- Function without Parameters and with Return value
- Function with Parameters and with Return value

### ***Function without Parameters and without Return value***

In this type of functions, there is no data transfer between calling function and called function. Simply the execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function.

***//Program to find using simple interest using functions.***

```
#include<stdio.h>
#include<conio.h>
void si() //function definition without parameters & without return value
{
    int year,period;
    float rate,sum,prin;
    clrscr();
    printf("Enter principle amt,rate & period\n");
    scanf("%f%f%d",&prin,&rate,&period);
    sum=prin;
    year=1;
    while(year<=period)
    {
        sum=sum*(1+rate);
        year++;
    }
    printf("Principal amt=%.2f\n",prin);
    printf("Interest rate=%.2f\n",rate);
    printf("Time period=%d\n",period);
    printf("Amt for %d years=%.2f with interest\n",period,sum);
}
void main()
{
    clrscr();
    si(); //function call
    getch();
}
```

---

**-----OUTPUT-----**

Enter principle amt, rate & period 5000 0.5 2

Principal Amt=5000.00

Interest rate=0.50

Time period=2

Amt for 2 years=11250.00 with interest

***Function with Parameters and without Return value***

In this type of functions there is data transfer from calling function to called function (parameters) but there is no data transfer from called function to calling function (return value). The execution control jumps from calling function to called function along with the parameters and executes called function, and finally comes back to the calling function. For example, consider the following program.

```
/*Program to read 3 sides of triangle and find the area and parameter of triangle using functions. */
```

```
#include<stdio.h>  
#include<conio.h>  
#include<math.h>  
void triangle(int a,int b,int c) //function definition  
{  
    float s,area;  
    s=(a+b+c)/2.0;  
    area=sqrt(s*((s-a)*(s-b)*(s-c)));  
    printf("The area of triangle=%.2f\n",area);  
    if((a==b)&&(b==c))  
        printf("The given triangle is equilateral\n");  
    else if((a==b) || (b==c) || (c==a))  
        printf("The given triangle is isosceles\n");  
    else  
        printf("The given triangle is scalene\n");  
}  
void main()  
{  
    int a,b,c;  
    clrscr();  
    printf("Enter the 3 sides of triangle\n");  
    scanf("%d%d%d",&a,&b,&c);  
    triangle(a,b,c); //function Call  
    getch();  
}
```

---

**-----OUTPUT-----**

Enter the 3 sides of triangle 6 6 6

The area of triangle=15.59

The given triangle is equilateral

Enter the 3 sides of triangle 5 5 4

The area of triangle=9.17

The given triangle is isosceles

Enter the 3 sides of triangle 4 5 6

The area of triangle=9.92

The given triangle is scalene

***Function without Parameters and with Return value***

In this type of functions there is no data transfer from calling function to called function (parameters) but there is data transfer from called function to calling function (return value). The execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function along with a return value. For example, consider the following program...

```
//Program to addition of two numbers using functions.
```

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

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

```
int addition() ;//function declaration
```

```
void main()
```

```
{
```

```
    int result ;
```

```
    clrscr() ;
```

```
    result = addition() ;//function call
```

```
    printf("Sum = %d", result) ;
```

```
    getch() ;
```

```
}
```

```
int addition() // function definition
```

```
{
```

```
    int num1, num2 ;
```

```
    printf("Enter any two integer numbers : ") ;
```

```
    scanf("%d%d", &num1, &num2);
```

```
    return (num1+num2) ;
```

```
}
```

---

**-----OUTPUT-----**

---

```
Enter any two integer numbers: 22 55
```

```
Sum = 77
```

***Function with Parameters and with Return value***

In this type of functions there is data transfer from calling function to called function (parameters) and also from called function to calling function (return value). The execution control jumps from calling function to called function along with parameters and executes called function, and finally comes back to the calling function along with a return value. For example, consider the following program.

```
//Program to addition of two numbers using functions.
```

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

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

```
int addition(int, int) ;//function declaration
```

```
void main()
```

```
{
```

```
    int num1, num2, result ;
```

```
    clrscr() ;
```

```
    printf("Enter any two integer numbers : ") ;
```

```
scanf("%d%d", &num1, &num2);
result = addition(num1, num2) ;//function call
printf("Sum = %d", result) ;
getch() ;
}
int addition(int a, int b) //function definition
{
    return (a+b) ;
}
```

-----OUTPUT-----

Enter any two-integer numbers: 22 55

Sum = 77

### Passing Argument to Function:

1. In C Programming, we have different ways of parameter passing schemes such as Call by Value and Call by Reference.
2. Function is good programming style in which we can write reusable code that can be called whenever require.
3. Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called **argument**.

### Two Ways of Passing Argument to Function in C Language:

1. Call by Value
2. Call by Reference

#### Call by Value

1. While Passing Parameters using call by value, Xerox copy of original parameter is created and passed to the called function.
2. Any update made inside method will not affect the original value of variable in calling function.
3. In the below example, **a** and **b** are the original values, Xerox copy of these values is passed to the function, and these values are copied into **a**, **b** variable of sum function respectively.
4. As their scope is limited to only function so, they cannot alter the values inside main function.

**// Program to swap two integer values using function (call by value).**

```
#include<stdio.h>
#include<conio.h>
void swap(int a,int b)
{
    int temp;
    temp=a;
    a=b;
```



```

    b=temp;
    printf("Values inside the function a=%d and b=%d\n",a,b);
}

void main()
{
    int a,b;
    clrscr();
    printf("Enter any two no's\n");
    scanf("%d%d",&a,&b);
    printf("Before swapping a=%d and b=%d\n",a,b);
    swap(a,b);
    printf("After swapping a=%d and b=%d\n",a,b);
    getch();
}

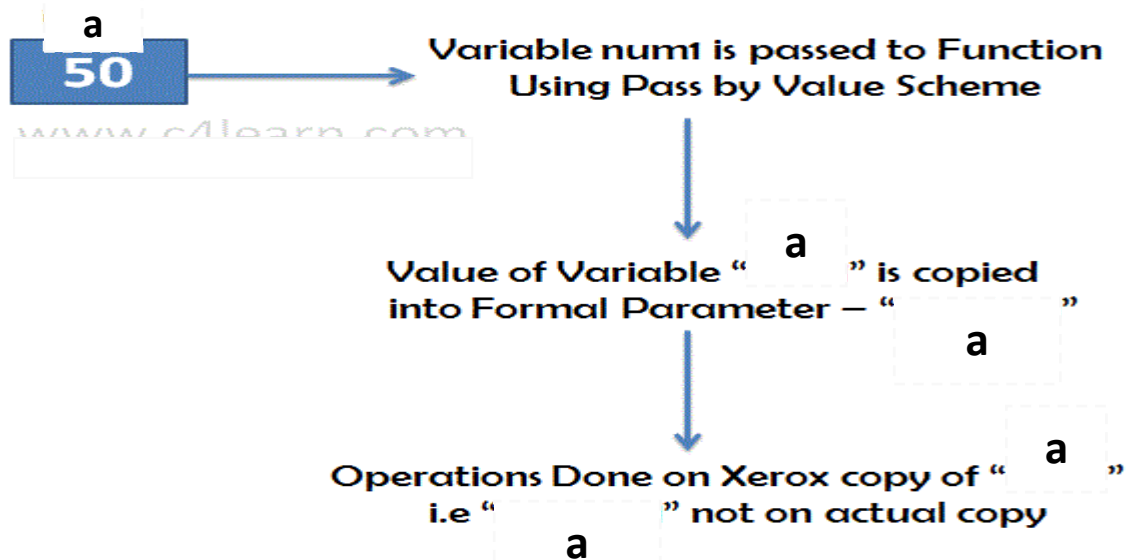
```

**-----OUTPUT-----**

```

Enter any two no's
5 2
Before swapping a=5 and b=2
Values inside the function a=2 and b=5
After swapping a=5 and b=2

```



**Parameter**

In computer programming, a parameter is a special kind of variable, used in a subroutine to refer to one of the pieces of data provided as input to the subroutine. These pieces of data are called *arguments*.

**Note:**

- Parameter Means Values Supplied to Function so that Function can Utilize These Values.
- Parameters are Simply Variables.
- Difference between Normal Variable and Parameter is that "These Arguments are defined at the time of Calling Function".

- Syntactically we can pass any number of parameter to function.
- Parameters are specified within Pair of Parenthesis.
- These Parameters are Separated by Comma (,)

**Example:** display (a,b,c,d,e);

**Parameter:** The names given in the function definition are called Parameters.

**Argument:** The values supplied in the function call are called Arguments.

### Types of Parameters or Arguments

#### 1. Formal Parameter :

Parameter Written In Function Definition is Called "**Formal Parameter**".

```
void main()
{
int num1;
display(num1);
}
```

```
void display(int para1)
{
-----
-----
}
```

Para1 is "Formal Parameter"

#### 2. Actual Parameter :

Parameter Written In Function Call is Called "**Actual Parameter**".

```
void main()
{
int num1;
display(num1);
}
```

```
void display(int para1)
{
-----
-----
}
```

num1 is "Actual Parameter"

Actual Parameter	Formal Parameter
1. It is written at the time of function calling.	1. It is written at the time of function definition.
2. It may be a variable or constant.	2. It must be a variable.
3. Example:- <pre>void main() {     int a=2, b=3;     sum(30,40);     sum(a,b); }</pre> <p>Here, 30 &amp; 40 and a &amp; b are Actual Parameter.</p>	4. <pre>void sum(int a, int b) {     .     . }</pre> <p>Here, a and b are Formal Parameters.</p>

### Call by Address or Reference method

1. While passing parameter-using call by address scheme, we are passing the actual address of the variable to the called function.
2. Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.

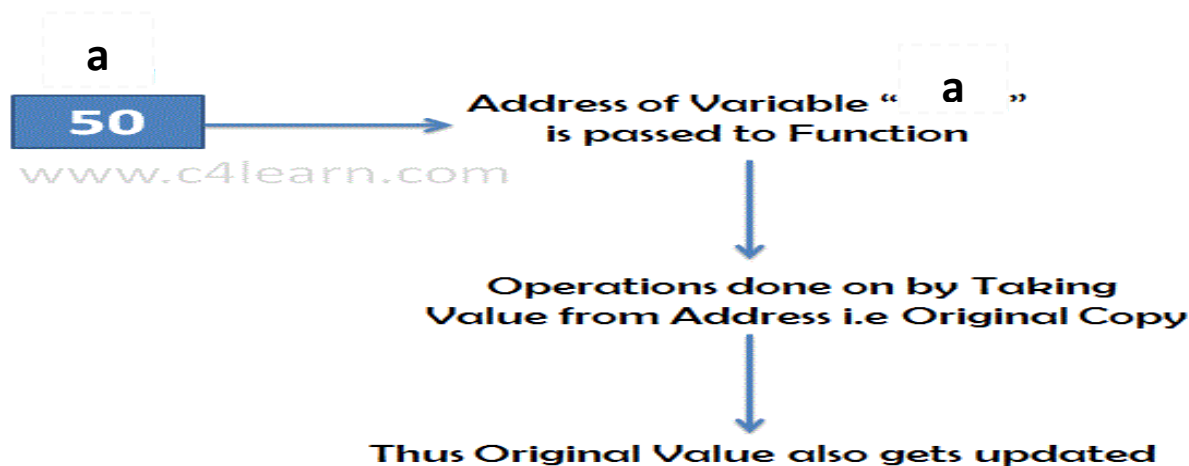
**//Program to swap two integer values using pointer and functions (Call by reference).**

```
#include<stdio.h>
void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
    printf("Values inside the function a=%d and b=%d\n",*a,*b);
}

void main()
{
    int a,b;
    printf("Enter any two numbers\n");
    scanf("%d%d",&a,&b);
    printf("Before swapping a=%d and b=%d\n",a,b);
    swap(&a,&b);
    printf("After swapping a=%d and b=%d\n",a,b);
    getch();
}
```

### **-----OUTPUT-----**

```
Enter any two numbers 9 5
Before swapping a=9 and b=5
Values inside the function a=5 and b=9
After swapping a=5 and b=9
```



**Comparison between Call by Value and Call by Reference in Programming:**

	<b>Call by Value</b>	<b>Call by Reference</b>
Description	A function to pass data or value to other functions	A function to pass data or value to other functions
Languages used	C based programming languages	C based programming languages such as C++, Java, but not C itself
Purpose	To pass arguments to another function	To pass arguments to another function
Arguments	A copy of actual arguments is passed to respective formal arguments	Reference the location or address of the actual arguments is passed to the formal arguments
Changes	Changes are made in the own personal copy. Changes made inside the function are not reflected on other functions	Any changes made in the formal arguments will also reflect in the actual arguments. Changes made inside the function are reflected outside the function as well.
Value modification	Original value is not modified.	Original value is modified.
Memory Location	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location
Safety	Actual arguments remain safe, they cannot be modified accidentally.	Actual arguments are not safe. They can be accidentally modified. Hence care is required when handling arguments.
Default	Default in most programming languages such as C++, PHP, Visual Basic .NET, C# and REAL basic	Supported by most programming languages, but not as default.

**Summary of Call By Value and Call By Reference:**

Point	Call by Value	Call by Reference
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function

**Scope of Variables**

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable, it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called *local variables*.
- Outside of all functions, which is called *global variables*.
- In the definition of function parameters which are called *formal parameters*.

**Local Variables**

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

**Global Variables**

Global variables are defined outside a function or above the main ( ) function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

**Recursive Function in C**

When Function is call within same function is called Recursion. The function, which call same function, is called recursive function. In other word when a function call itself then that, function is called Recursive function.

Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

**Advantage of Recursion**

- Function calling related information will be maintained by recursion.
- Stack evaluation will be take place by using recursion.
- In fix prefix, post-fix notation will be evaluated by using recursion.

Disadvantage of Recursion

- It is a very slow process due to stack overlapping.
- Recursive programs can create stack overflow.
- Recursive functions can create as loops.

**//Program to find factorial of a given number using recursive functions.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    int fact(int); //function prototype
    clrscr();
    printf("Enter any number\n");
    scanf("%d",&n);
    if(n<0)
    printf("Invalid input enter any positive number\n");
    else
    printf("Factorial of %d is %d\n",n,fact(n));
    getch();
}
```

**//Recursive function to find the factorial of given number**

```
int fact(int n)
{
    if(n==0)
    return(1);
    else
    return (n*fact(n-1));
}
```

## -----OUTPUT-----

Enter any number 5

Factorial of 5 is 120

**return(n \* fact( n - 1));**

1 step → **return(5 \* fact (4)) = 120**

2 step → **return(4 \* fact (3)) = 24**

3 step → **return(3 \* fact (2)) = 6**

4 step → **return(2 \* fact (1)) = 2**

5 step → **return(1 \* fact (0)) = 1**

**factorial 5 = 5\*4\*3\*2\*1**

**factorial 4 = 4\*3\*2\*1**

**factorial 3 = 3\*2\*1**

**factorial 2 = 2\*1**

**factorial 1 = 1\*1**



## STRUCTURE

Structure is a collection of different data types which are grouped together. Structure is set of heterogeneous elements. Each element in a C structure is called member.

Let's say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. This may sound confusing, do not worry we will understand this with the help of example.

- If you want to access structure members in C, structure variable should be declared.
- Many structure variables can be declared for same structure and memory will be allocated for each separately.
- It is a best practice to initialize a structure to null while declaring, if we don't assign any values to structure members.

## DIFFERENCE BETWEEN C VARIABLE, C ARRAY AND C STRUCTURE:

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types and Data types can be int, char, float, double and long double etc.

## DECLARATION OF STRUCTURE

We use **struct** keyword to create a **structure** in C. The **struct** keyword is a short form of **structured data type**.

```
struct struct_name
```

```
{
```

```
    DataType member1_name;
```

```
    DataType member2_name;
```

```
    DataType member3_name;
```

```
    ...
```

```
};
```

Here struct\_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

First we will see the syntax of creating struct variable, accessing struct members etc and then we will see a complete example.



### Example of Structure

```
struct Book
{
    char name[15];
    int price;
    int pages;
};
```

Here the **struct Book** declares a structure to hold the details of book which consists of three data fields, namely *name*, *price* and *pages*. These fields are called **structure elements or members**. Each member can have different data type, like in this case, **name** is of char type and **price** is of int type etc. **Book** is the name of the structure and is called structure tag.

### DECLARE VARIABLE OF A STRUCTURE

It is possible to declare variables of a structure, after the structure is defined. Structure variable declaration is similar to the declaration of variables of any other data types. Structure variables can be declared in following two ways.

#### 1) Declaring Structure variables separately

```
struct struct_name var_name;
```

Example:

```
struct Student
{
    char[20] name;
    int age;
    int rollno;
};
```

```
struct Student S1, S2; //declaring variables of Student
```

#### 2) Declaring Structure Variables with Structure definition

```
struct struct_name
{
    DataType member1_name;
    DataType member2_name;
    DataType member3_name;
} var_name;
```

Example:

```
struct Student
{
    char[20] name;
    int age;
    int rollno;
} S1, S2 ;
```

Here **S1** and **S2** are variables of structure **Student**. However, this approach is not much recommended.

### ACCESS DATA MEMBERS OF A STRUCTURE USING A STRUCT VARIABLE

Structure members can be accessed and assigned values in number of ways. Structure member has no meaning independently. In order to assign a value to a structure member, the member name must be linked with the **structure** variable using dot. operator also called **period** or **member access** operator.

```
var_name.member1_name;  
var_name.member2_name;
```

#### **Example:**

```
struct Book  
{  
    char name[15];  
    int price;  
    int pages;  
} b1 , b2 ;
```

b1.price=200;    //b1 is variable of Book type and price is member of Book

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", b1.name);  
scanf(" %d ", &b1.price);
```

### ASSIGN OR INITIALIZATION OF VALUES TO STRUCTURE MEMBERS

There are three ways to do this.

#### 1) Using Dot(.) operator

```
var_name.memeber_name = value;
```

#### **Example:**

```
struct patient p1;  
p1.height = 180.75;    //initialization of each member separately  
p1.weight = 73;  
p1.age = 23;
```

#### 2) All members assigned in one statement

```
struct struct_name var_name = {value for memeber1, value for memeber2 ...so on  
                                for all the members}
```

#### **Example:**

```
struct Patient  
{ float height;  
  int weight;  
  int age; };  
struct Patient p1 = { 180.75 , 73, 23 };    //initialization
```

### **ARRAY OF STRUCTURE**

Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object. As we know, an array is a collection of similar type, therefore an array can be of structure type.

### **SYNTAX FOR DECLARING STRUCTURE ARRAY**

```
struct struct-name
{
    datatype var1;
    datatype var2;
    -----
    -----
    datatype varN;
};

struct struct-name obj [ size ];
```

### **ARRAY WITHIN STRUCTURE**

As we know, structure is collection of different data type. Like normal data type, it can also store an array as well.

### **SYNTAX FOR ARRAY WITHIN STRUCTURE**

```
struct struct-name
{
    datatype var1;           // normal variable
    datatype array [size];  // array variable
    -----
    -----
    datatype varN;
};

struct struct-name obj;
```

### **POINTER TO STRUCTURE**

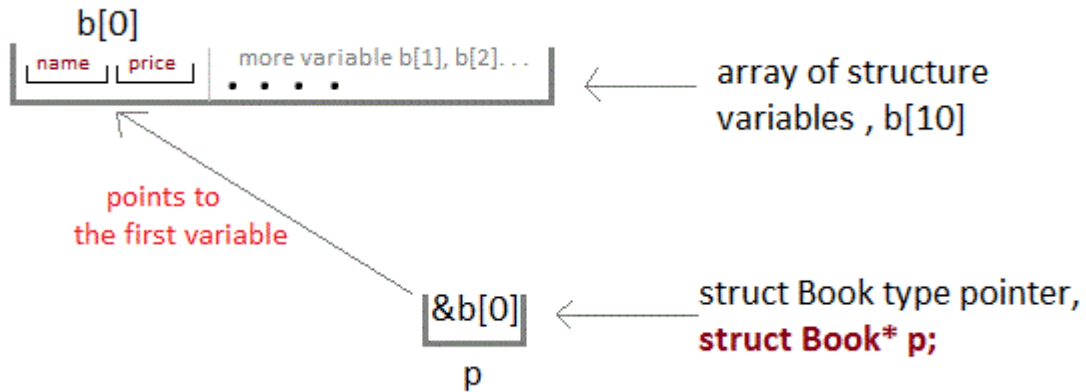
Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to make the use of array of structure variables efficiently, we use pointers of structure type. We can also have pointer to a single structure variable, but it is mostly used with array of structure variables.

```
struct Book
{
    char name[10];
    int price;
}

void main()
{
```

```
struct Book a;    //Single structure variable
struct Book* ptr; //Pointer of Structure type
ptr = &a;

struct Book b[10]; //Array of structure variables
struct Book* p;    //Pointer of Structure type
p = &b;
}
```



### ACCESSING STRUCTURE MEMBERS WITH POINTER

To access members of structure with structure variable, we used the dot operator. But when we have a pointer of structure type, we use arrow `->` to access structure members.

```
#include <stdio.h>
void main()
{
    struct my_structure {
        char name[20];
        int number;
        int rank;
    };

    struct my_structure variable = {"StudyTonight",35,1};
    struct my_structure *ptr;
    ptr = &variable;
    printf("NAME: %s\n",ptr->name);
    printf("NUMBER: %d\n",ptr->number);
    printf("RANK: %d",ptr->rank);
}
```

### OUTPUT:

```
NAME: StudyTonight
NUMBER: 35
RANK: 1
```

**UNION**

C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members
- We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Whereas Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.
- Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Using normal variable	Using pointer variable
<b>Syntax:</b> <pre>union tag_name { data type var_name1; data type var_name2; data type var_name3; };</pre>	<b>Syntax:</b> <pre>union tag_name { data type var_name1; data type var_name2; data type var_name3; };</pre>
<b>Example:</b> <pre>union student { int mark; char name[10]; float average; };</pre>	<b>Example:</b> <pre>union student { int mark; char name[10]; float average; };</pre>
<b>Declaring union using normal variable:</b> <pre>union student report;</pre>	<b>Declaring union using pointer variable:</b> <pre>union student *report, rep;</pre>
<b>Initializing union using normal variable:</b> <pre>union student report = {100,</pre>	<b>Initializing union using pointer variable:</b> <pre>union student rep = {100,</pre>



"Mani", 99.5};	"Mani", 99.5}; report = &rep;
<b>Accessing union members using normal variable:</b> report.mark; report.name; report.average;	<b>Accessing union members using pointer variable:</b> report -> mark; report -> name; report -> average;

Structure	Union
1. The keyword <b>struct</b> is used to define a structure	1. The keyword union is used to define a union.
2. When a <b>variable</b> is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by <b>individual</b> members of union.
4. The address of each member will be in <b>ascending order</b> . This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5. Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once.	7. Only the first member of a union can be initialized.

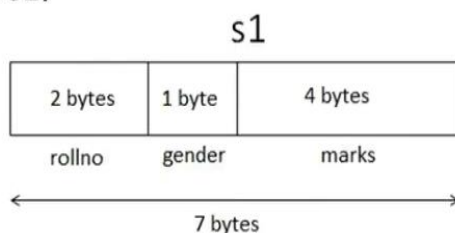
```
union student {
```

```
    int rollno;
```

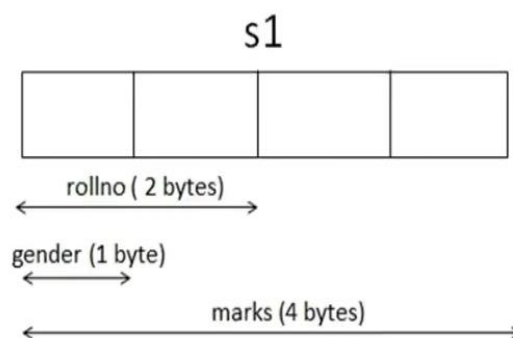
```
    char gender;
```

```
    float marks;
```

```
} s1;
```



Memory Allocation in Structure



Memory Allocation in Union

## POINTERS

A Pointer is a variable that holds address of another variable of same data type. Unlike other variables that hold values of a certain type, pointer holds the address of a variable.

For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of an integer variable. In this guide, we will discuss pointers in C programming with the help of examples.

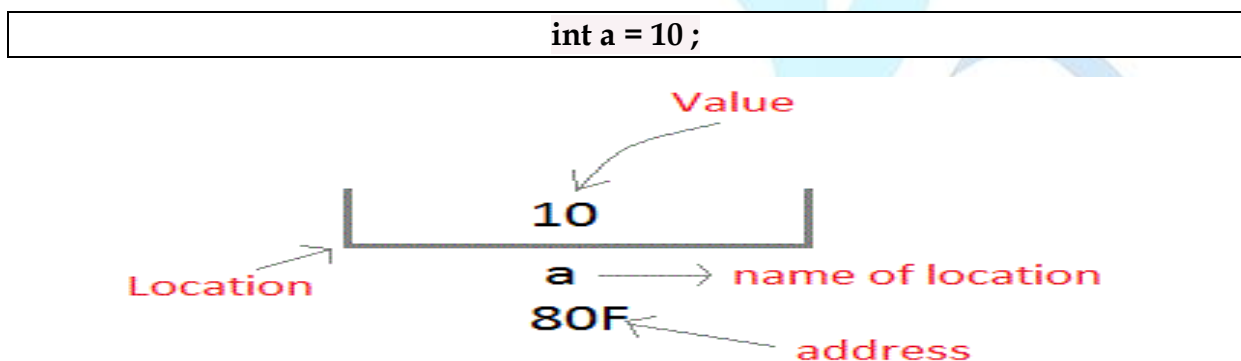
### Benefits of using pointers

- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It reduces length of the program and its execution time.
- It allows C to support dynamic memory management.

### Concept of Pointer

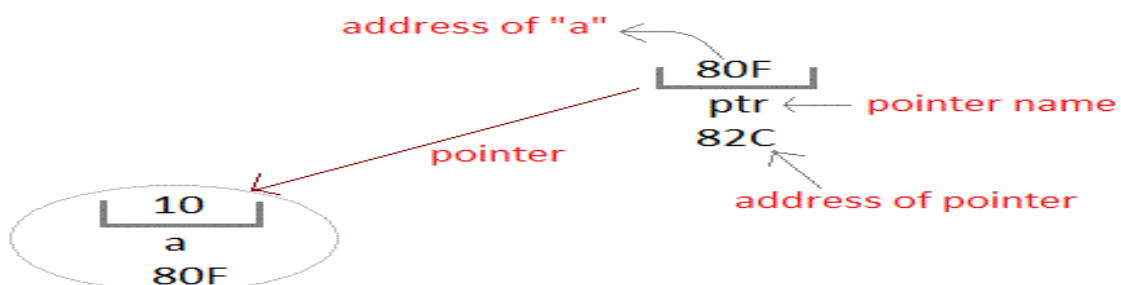
Whenever a variable is declared in the program, system allocates a location i.e. an address to that variable in the memory, to hold the assigned value. This location has its own address number.

Let us assume that system has allocated memory location 80F for a variable a.



We can access the value 10 by either using the variable name a or the address 80F. Since the memory addresses are simply numbers they can be assigned to some other variable. The variable that holds memory address are called pointer variables.

A **pointer** variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of pointer variable will be stored in another memory location.





## DECLARING A POINTER VARIABLE

General syntax of pointer declaration is,

<b>data-type *pointer_name;</b>
---------------------------------

Data type of a pointer must be same as the data type of a variable to which the pointer variable is pointing. void type pointer works with all data types, but is not used often used.

### **Initialization of Pointer variable**

Pointer Initialization is the process of assigning address of a variable to pointer variable. Pointer variable contains address of variable of same data type. In C language address **operator &** is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

<pre>int a = 10 ; int *ptr ;    //pointer declaration ptr = &amp;a ;    //pointer initialization or, int *ptr = &amp;a ; //initialization and declaration together</pre>
--

Pointer variable always points to same type of data.

<pre>float a; int *ptr; ptr = &amp;a; //ERROR, type mismatch</pre>
--

**Note:** If you do not have an exact address to be assigned to a pointer variable while declaration, It is recommended to assign a NULL value to the pointer variable. A pointer which is assigned a NULL value is called a null pointer.

### **Dereferencing of Pointer**

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using **the indirection operator \***.

<pre>int a,*p; a = 10; p = &amp;a; printf("%d",*p); //this will print the value of a. printf("%d",&amp;a); //this will also print the value of a. printf("%u",&amp;a); //this will print the address of a. printf("%u",p); //this will also print the address of a. printf("%u",&amp;p); //this will print the address of p.</pre>
--

**Points to remember:**

1. While declaring/initializing the pointer variable, \* indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand '&'.
3. The pointer variable stores the address of a variable. The declaration `int *a` doesn't mean that `a` is going to contain an integer value. It means that `a` is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, \* is used. Here, the \* can be read as '**value at**'.

```
/*Program to show the declaration of the pointer and access the values and display its value. */
#include<stdio.h>
#include<conio.h>
void main()
{
int i,*iptr;
float f1,*fptr;
clrscr();
printf("Enter the integer\n");
scanf("%d",&i);
printf("Enter the floating no\n");
scanf("%f",&f1);
iptr=&i;
fptr=&f1;
printf("\n Value of i=%d\n",i);
printf("\n Address of i=%u\n",iptr);
printf("\n Value of f=%f\n",f1);
printf("\n Address of f=%u\n",fptr);
getch();
}
```

-----**OUTPUT**-----

Enter the integer 123  
Enter the floating no 147.4

Value of i=123  
Address of i=65524

Value of f=147.4  
Address of f=65518

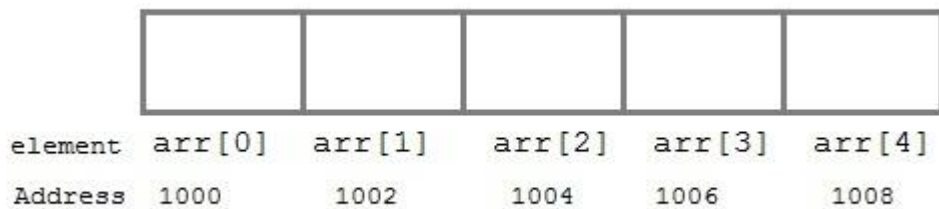
## **POINTER AND ARRAYS**

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e. address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

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

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of an array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] //by default

We can declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
or p = &arr[0]; //both the statements are equivalent.
```

Now we can access every element of array arr using p++ to move from one element to another.

**NOTE:** You cannot decrement a pointer once incremented. p-- won't work.

### **Pointer to Array**

As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array. Lets have an example,

```
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a; // same as int*p = &a[0]
for (i=0; i<5; i++)
{
    printf("%d", *p);
    p++;
}
```

In the above program, the pointer \*p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as pointer and print all the values.

Replacing the **printf("%d", \*p);** statement of above example, with below mentioned statements. Lets see what will be the result.

**printf("%d", a[i]);** → **prints the array, by incrementing index**

**printf("%d", i[a] );** → **this will also print elements of array**

**printf("%d", a+i );** → **This will print address of all the array elements**

**printf("%d", \*(a+i) );** → **Will print value of array element.**

**printf("%d", \*a);** → **will print value of a[0] only**

**a++;** → **Compile time error, we cannot change base address of the array.**

The generalized form for using pointer with an array,

<b>*(a+i)</b>
---------------

is same as:

<b>a[i]</b>
-------------

### **Pointer to Multidimensional Array**

A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0] element.

Here is the generalized form for using pointer with multidimensional arrays.

<b>*(a + i + j)</b>
---------------------

is same as

<b>a[i][j]</b>
----------------

### **Pointer and Character strings**

Pointer can also be used to create strings. Pointer variables of char type are treated as string.

<b>char *str = "Hello";</b>
-----------------------------

This creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello". Another important thing to note that string created using charpointer can be assigned a value at runtime.

```
char *str;
str = "hello"; //this is Legal
```

The content of the string can be printed using printf() and puts().

```
printf("%s", str);
puts(str);
```

Notice that str is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator \*.

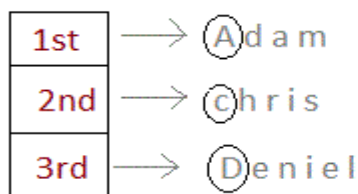
### Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {
    "Adam",
    "chris",
    "Deniel"
};

//Now see same array without using pointer
char name[3][20] = {
    "Adam",
    "chris",
    "Deniel"
};
```

#### Using Pointer



char\* name[3]

**Only 3 locations for pointers, which will point to the first character of their respective strings.**

#### Without Pointer

A	d	a	m			
c	h	r	i	s		
D	e	n	i	e	l	

char name[3][20]

**extends till 20 memory locations**

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

**POINTER TO A POINTER**

Pointers are used to store the address of the variables of specified datatype. But If you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as Pointer to Pointer variable.

Syntax:

<b>int **p1;</b>
------------------

Here, we have used two indirection operator (\*) that stores and points to the address of a pointer variable i.e, int \*. If we want to store the address of this variable p1, then the syntax would be:

<b>int ***p2</b>
------------------

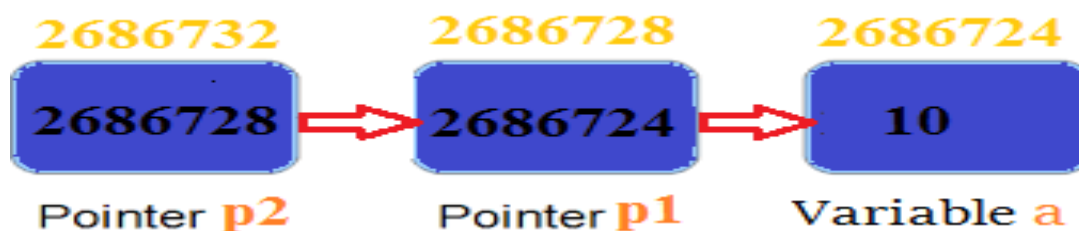
Hence, number of indirection operator (\*) - 1 tells you to what type will the current pointer variable will point to.

**SIMPLE PROGRAM TO REPRESENT POINTER TO A POINTER**

```
#include <stdio.h>
void main()
{
    int a=10;
    int *p1; //this can store the address of variable a
    int **p2;
    /*this can store the address of pointer variable p1 only. It cannot store the
    address of variable a */
    p1 = &a;
    p2 = &p1;
    printf("Address of a = %u\n",&a);
    printf("Address of p1 = %u\n",&p1);
    printf("Address of p2 = %u\n\n",&p2);
    printf("Value at the address stored by p2 = %u\n",*p2); //this will give the
address of a
    printf("Value at the address stored by p1 = %d\n\n",*p1);
    printf("Value of **p2 = %d\n", **p2); //read this *(p2)
    /*This is not allowed, it will give a compiler time error
    p2 = &a;
    printf("%u",p2);*/
}
```

**OUTPUT:**

```
Address of a = 2686724
Address of p1 = 2686728
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
Value of **p2 = 10
```

Explanation of the above program

- **p1** pointer variable can only hold the address of the variable a.(i.e Number of indirection operator(\*)-1 variable). Similarly, **p2** variable can only hold the address of variable **p1**. It cannot hold the address of variable a.
- **\*p2** gives us the value at an address stored by the **p2** pointer. **p2** stores the address of **p1** pointer and value at the address of **p1** is the address of variable **a**. Thus, **\*p2** prints address of **a**.
- **\*\*p2** can be read as **\*( \*p2)**. Hence, it gives us the value stored at the address **\*p2**. From above statement, you know **\*p2** means the address of variable a. Hence, the value at the address **\*p2** is 10. Thus, **\*\*p2** prints 10.

FILE HANDLING IN C LANGUAGE

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure.

In C language, we use a structure pointer of file type to declare a file.

```
FILE *fp;
```

C provides a number of functions that helps to perform basic file operations. Following are the functions,

FUNCTION	DESCRIPTION
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point



ftell()	gives current position in the file
rewind()	set the position to the beginning point

**Opening a File or Creating a File**

The fopen() function is used to create a new file or to open an existing file.

**General Syntax:**

<b>*fp = FILE *fopen(const char *filename, const char *mode);</b>
---

Here filename is the name of the file to be opened and mode specifies the purpose of opening the file. Mode can be of following types,

**\*fp** is the FILE pointer (**FILE \*fp**), which will hold the reference to the opened (or created) file.

MODE	DESCRIPTION
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

### **Closing a File**

The fclose() function is used to close an already opened file.

#### **General Syntax :**

<b>int fclose( FILE *fp );</b>
--------------------------------

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file. This EOF is a constant defined in the header file stdio.h.

### **Input/output operation on File**

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are simplest functions used to read and write individual characters to a file.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    char ch;
    fp = fopen("one.txt", "w");
    printf("Enter data");
    while( (ch = getchar()) != EOF) {
        putc(ch,fp);
    }
    fclose(fp);
    fp = fopen("one.txt", "r");
    while( (ch = getc(fp)) != EOF)
        printf("%c",ch);
    fclose(fp);
}
```

### **Reading and Writing from File using fprintf() and fscanf()**

```
#include<stdio.h>
#include<conio.h>
struct emp
{
    char name[10];
    int age;
};

void main()
{
    struct emp e;
```

```

FILE *p,*q;
p = fopen("one.txt", "a");
q = fopen("one.txt", "r");
printf("Enter Name and Age");
scanf("%s %d", e.name, &e.age);
fprintf(p,"%s %d", e.name, e.age);
fclose(p);
do
{
    fscanf(q,"%s %d", e.name, e.age);
    printf("%s %d", e.name, e.age);
}
while( !feof(q) );
getch();
}

```

In this program, we have created two FILE pointers and both are referring to the same file but in different modes. fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on console using standard printf() function.

### **Difference between Append and Write Mode**

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exist already.

The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file (if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

### **C PREPROCESSOR DIRECTIVES or MACROS:**

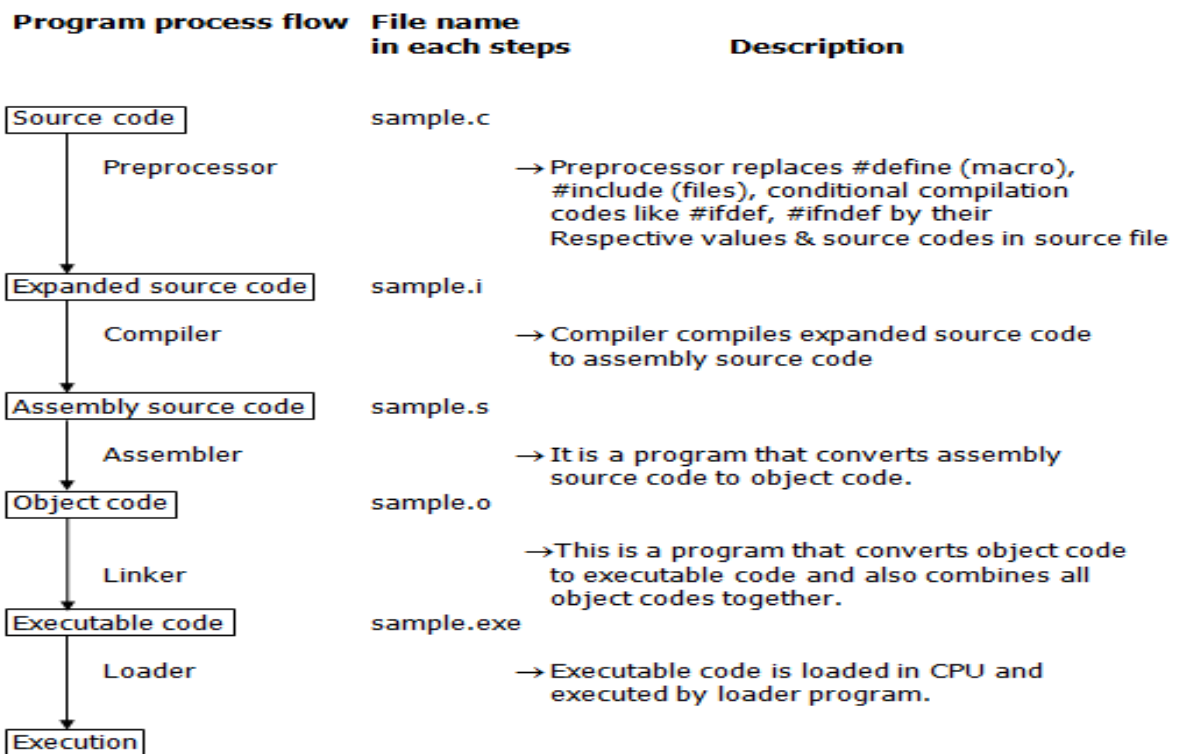
- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.

Below is the list of preprocessor directives that C programming language offers.

Pre-processor	Syntax/Description
Macro	<b>Syntax:</b> #define This macro defines constant value and can be any of the basic data types.
Header file	<b>Syntax:</b> #include<file_name>

inclusion	The source code of the file "file_name" is included in the main program at the specified place.
Conditional compilation	<b>Syntax:</b> #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	<b>Syntax:</b> #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.



### Including Header Files

The #include preprocessor is used to include header files to a C program. For example,

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

Here, "stdio.h" is a header file. The #include preprocessor directive replaces the above line with the contents of stdio.h header file which contains function and macro definitions.

That's the reason why you need to use #include <stdio.h> before you can use functions like scanf() and printf().

You can also create your own header file containing function declaration and include it in your program using this preprocessor directive.

**#include "my\_header.h"**

### **Macros using #define**

You can define a macro in C using #define preprocessor directive.

A macro is a fragment of code that is given a name. You can use that fragment of code in your program by using the name. For example,

**#define c 299792458 // speed of light**

Here, when we use c in our program, it's replaced by 3.1415.

You can also define macros that works like a function call, known as function-like macros. For example,

**#define circleArea(r) (3.1415\*r\*r)**

Every time the program encounters circleArea(argument), it is replaced by (3.1415\*(argument)\*(argument)).

Suppose, we passed 5 as an argument then, it expands as below:

**circleArea(5) expands to (3.1415\*5\*5)**

### **Conditional Compilation**

In C programming, you can instruct preprocessor whether to include certain chunk of code or not. To do so, conditional directives can be used.

It's similar like a if statement. However, there is a big difference you need to understand.

The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals is used to include (or skip) certain chunks of code in your program before execution.

### **Uses of Conditional**

- use different code depending on the machine, operating system
- compile same source file in two different programs
- to exclude certain code from the program but to keep it as reference for future purpose

### **How to use conditional?**

To use conditional, #ifdef, #if, #defined, #else and #elseif directives are used.

## **#ifdef Directive**

```
#ifdef MACRO  
    conditional codes  
#endif
```

Here, the conditional codes are included in the program only if MACRO is defined.

## **#if, #elif and #else Directive**

```
#if expression  
    conditional codes  
#endif
```

Here, expression is a expression of integer type (can be integers, characters, arithmetic expression, macros and so on). The conditional codes are included in the program only if the expression is evaluated to a non-zero value.

The optional #else directive can have used with #if directive.

```
#if expression  
    conditional codes if expression is non-zero  
#else  
    conditional if expression is 0  
#endif
```

You can also add nested conditional to your #if...#else using #elif

```
#if expression  
    conditional codes if expression is non-zero  
#elif expression1  
    conditional codes if expression is non-zero  
#elif expression2  
    conditional codes if expression is non-zero  
... ..  
else  
    conditional if all expressions are 0  
#endif
```

## **#defined**

The special operator #defined is used to test whether certain macro is defined or not. It's often used with #if directive.

```
#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048  
    conditional codes
```