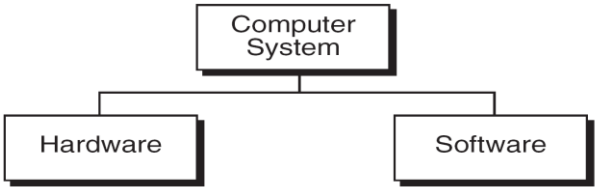
UNIT I

Computer systems:

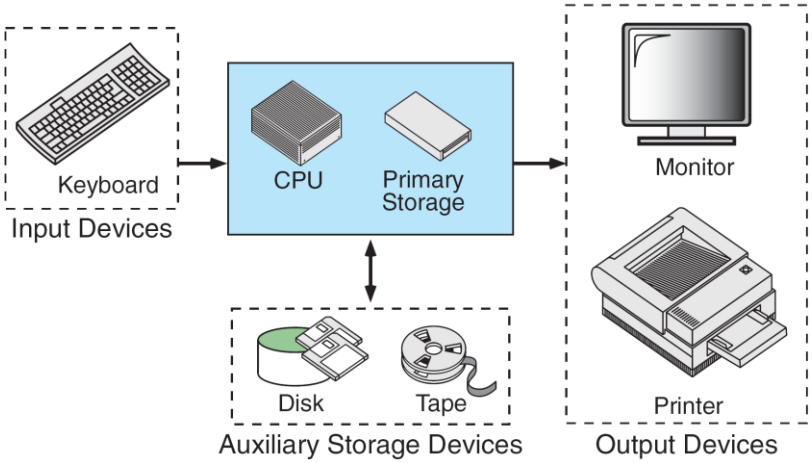
A Computer is an electronic device which performs operations such as accepts data
As an input, store the data, manipulate or process the data and produce the results an output.
Main task performed by a computer

- Accept the data
- Process or manipulate the data
- Display or store the result in the form of human understanding
- Store the data, instructions and results.

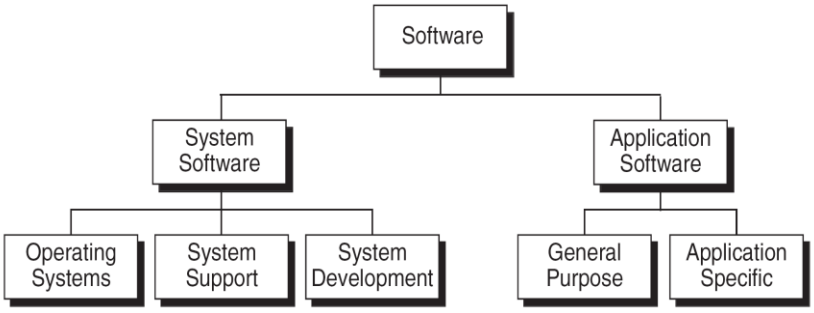


A computer system consists of hardware and software.

Computer hardware is the collection of physical elements that comprise a computer system.



Computer software is a collection of computer programs and related data that provides the instructions for a computer what to do and how to do it. Software refers to one or more computer programs and data held in the storage of the computer for some purpose



Basically computer software is of three main types

System Software: System software is responsible for managing a variety of independent hardware components, so that they can work together. Its purpose is to unburden the application software programmer from the often complex details of the particular computer being used, including such accessories as communications devices, printers, device readers, displays and keyboards, and also to partition the computer's resources such as memory and processor time in a safe and stable manner.

- Device drivers
- Operating systems
- Servers
- Utilities
- Window systems

Programming Software: Programming Software usually provides tools to assist a programmer in writing computer programs, and software using different programming languages in a more convenient way. The tools include:

- Compilers
- Debuggers
- Interpreters
- Linkers
- Text editors

Application Software: Application software is developed to aid in any task that benefits from computation. It is a broad category, and encompasses Software of many kinds, including the internet browser being used to display this page. This category includes:

- Business software
- Computer aided design
- Databases
- Decision making software
- Educational software
- Image editing

Computing Environment:

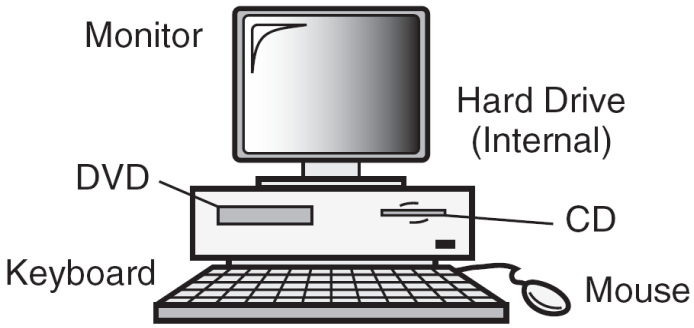
Computing Environment is a collection of computers / machines, software, and networks that support the processing and exchange of electronic information meant to support various types of computing solutions.

Types of Computing Environments:

- Personal Computing Environment

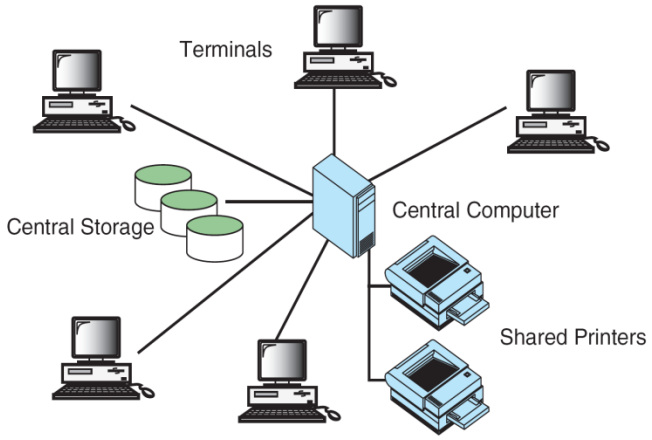
- Client Server Environment
- Time sharing Environment
- Distributed Environment

Personal Computing Environment:
All of the computer hardware components are tied together in our personal computer. A **personal computer** (**PC**) is a computer whose original sales price, size, and capabilities make it useful for individuals, and intended to be operated directly by an end user, with no intervening computer operator. People generally relate this term with Microsoft's Windows Operating system. Personal computers generally run on Windows, Mac or some version of Linux operating system.
Desktop: Desktop computer is just another version of Personal Computer intended for regular use from a single use. A computer that can be fit on a desk can also be called as desktop.



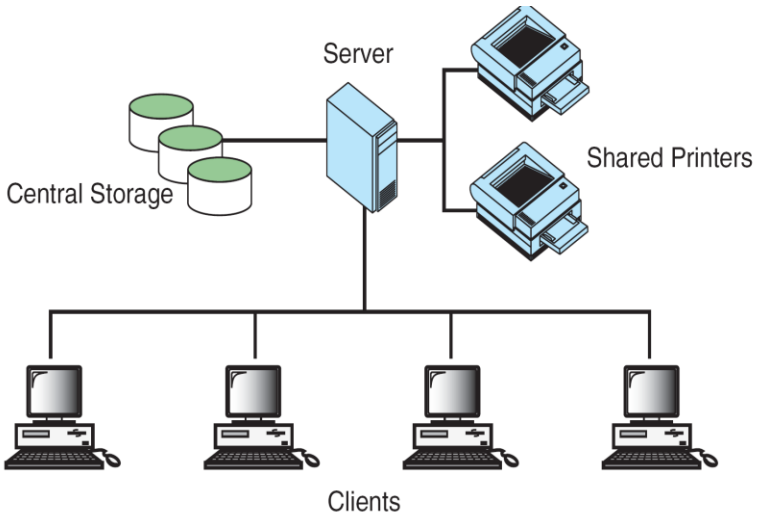
Time-Sharing Environment:

In the time-sharing environment, all computing must be done by the central computer. The central computer the shared resources, it manage the shared data and printing. Employees in large companies often work in what is known as time sharing environment. In the time sharing environment, many users are connected to one or more computers. These computers may be mini computers and central mainframes. In this environment the output devices, auxiliary storage devices are shared by all the users.



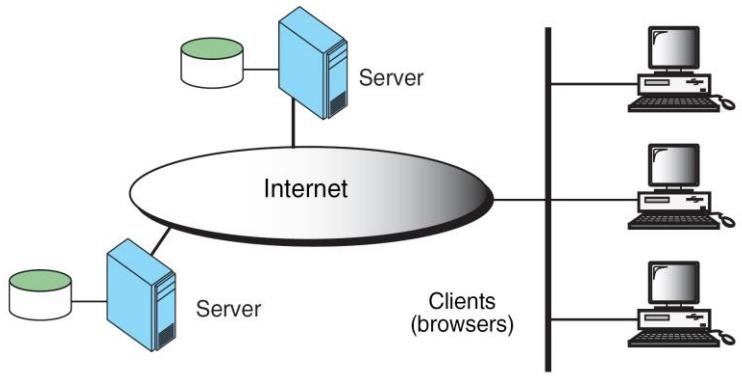
Client/Server Environment

Client/Server computing environment splits the computing function between a central computer and user's computers. The users are given personal computers or work stations so that some of the computation responsibility can be moved from the central computer and assigned to the workstations. In the client/server environment the users micro computers or work stations are called the client. The central computer which may be a powerful micro computer, minicomputer or central mainframe system is known as server.



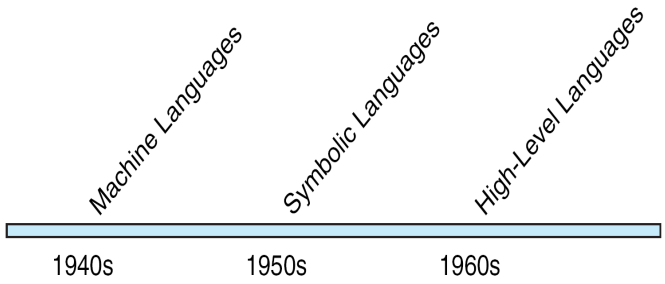
Distributed Computing Environment

A distributed computing environment provides a seamless integration of computing functions between different servers and clients. The internet provides connectivity to different servers throughout the world. This environment provides reliable, scalable and highly available network.



COMPUTER LANGUAGES

In order to communicate with the computer user also needs to have a language that should be understood by the computer. For this purpose, different languages are developed for performing different types of work on the computer. Basically, languages are divided into two categories according to their interpretation.



1. Low Level Languages.

2. High Level Languages.

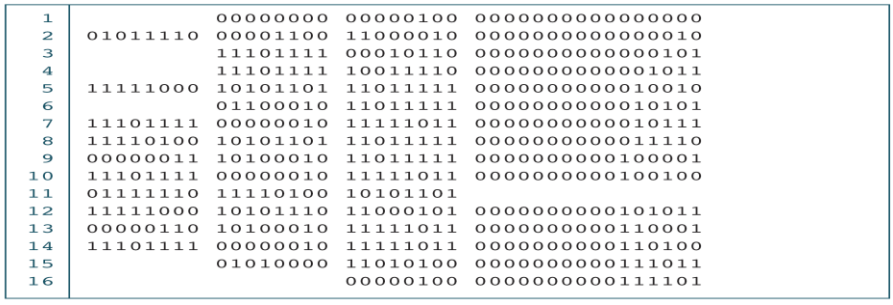
Low Level Languages

Low level computer languages are machine codes or close to it. Computer cannot understand instructions given in high level languages or in English. It can only understand and execute instructions given in the form of machine language i.e. language of 0 and 1. There are two types of low level languages:

- **Machine Language.**
- **Assembly Language**

Machine Language: It is the lowest and most elementary level of Programming language and was the first type of programming language to be Developed. Machine Language is basically the only language which computer Can understand. In fact, a manufacturer designs a computer to obey just one Language, its machine code, which is represented inside the computer by a String of binary digits (bits) 0 and 1. The symbol 0 stands for the absence of Electric pulse and 1 for the

presence of an electric pulse . Since a computer is Capable of recognizing electric signals, therefore, it understand machine Language.



Advantages of Machine Language

i) It makes fast and efficient use of the computer.

ii) It requires no translator to translate the code i.e. Directly understood by the computer

Disadvantages of Machine Language:

i) All operation codes have to be remembered

iv) These languages are machine dependent i.e. a particular

Machine language can be used on only one type of computer

Assembly Language

It was developed to overcome some of the many inconveniences of machine language. This is another low level but a very important language in which operation codes and operands are given in the form of alphanumeric symbols instead of 0's and l's. These alphanumeric symbols will be known as mnemonic codes and can have maximum up to 5 letter combination e.g. ADD for addition, SUB for subtraction, START,LABEL etc. Because of this feature it is also known as 'Symbolic Programming Language'. This language is also very difficult and needs a lot of practice to master it because very small

English support is given to this language. The language mainly helps in compiler orientations. The instructions of the Assembly language will also be converted to machine codes by language translator to be executed by the computer.

```
1       entry   main,^m<r2>
2       subl2   #12,sp
3       jsb     C$MAIN_ARGS
4       movab   $CHAR_STRING_CON
5
6       pushal  -8(fp)
7       pushal  (r2)
8       calls   #2,SCANF
9       pushal  -12(fp)
10      pushal  3(r2)
11      calls   #2,SCANF
12      mull3   -8(fp),-12(fp),-
13      pusha   6(r2)
14      calls   #2,PRINTF
15      clrl    r0
16      ret
```

Advantages of Assembly Language

i) It is easier to understand and use as compared to machine language.

ii) It is easy to locate and correct errors.

iii) It is modified easily

Disadvantages of Assembly Language

i) Like machine language it is also machine dependent.

ii) Since it is machine dependent therefore programmer Should have the knowledge of the hardware also.

High Level Languages

High level computer languages give formats close to English language and the purpose of developing high level languages is to enable people to write programs easily and in their own native language environment (English). High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes. Each instruction in the high level language is translated into many machine language instructions thus showing one-to-many translation

Types of High Level Languages

Many languages have been developed for achieving different variety of tasks, some are fairly specialized others are quite general purpose.

These are categorized according to their use as

a) Algebraic Formula-Type Processing. These languages are oriented towards the computational procedures for solving mathematical and statistical problem

Examples are

- **BASIC (Beginners All Purpose Symbolic Instruction Code).**
- **FORTRAN (Formula Translation).**
- **PL/I (Programming Language, Version 1).**
- **ALGOL (Algorithmic Language).**

- **APL (A Programming Language).**

b) Business Data Processing:

- These languages emphasize their capabilities for maintaining data processing procedures and files handling problems. Examples are:
- **COBOL (Common Business Oriented Language).**
- **RPG (Report Program Generator**

b) String and List Processing: These are used for string manipulation including search for patterns, inserting and deleting characters. Examples are:

- **LISP (List Processing).**
- **Prolog (Program in Logic).**

Object Oriented Programming Language

In OOP, the computer program is divided into objects. Examples are:

- **C++**
- **Java**

e) **Visual programming language**: these are designed for building Windows-based applications Examples are:

- **Visual Basic**
- **Visual Java**
- **Visual C**

```
 1  /* This program reads two integers from the keyboard
 2     and prints their product.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7
 8  int main (void)
 9  {
10  // Local Definitions
11     int number1;
12     int number2;
13     int result;
14
15  // Statements
16     scanf  ("%d", &number1);
```

```
17     scanf  ("%d", &number2);
18     result = number1 * number2;
19     printf ("%d", result);
20     return 0;
21  }  // main
```

Advantages of High Level Language

Following are the advantages of a high level language:

- User-friendly
- Similar to English with vocabulary of words and symbols
- Therefore it is easier to learn.
- They are easier to maintain.

Disadvantages of High Level Language

- A high-level language has to be translated into the machine language by a translator and thus a price in computer time is paid.
- The object code generated by a translator might be inefficient Compared to an equivalent assembly language program

Creating and Running Programs:

There are four steps in this process.

1. Writing and editing the program using Text editor (source code).
2. Compile the program using any C compiler.(.bak file)
3. Linking the program with the required library modules(object file)
4. Executing the program. (.Exe file)

Creating and Editing a C Program in C Programming Language compiler: Writing or creating and editing source program is a first step in c language. Source code is written in c programming language according to the type of problem or requirement, in any text editor.

Saving C Program in C Programming Language: Source code is saved on the secondary storage. Source code is saved as text file. The extension of file must be ".c". Example the file name is "learn c programming language.c"

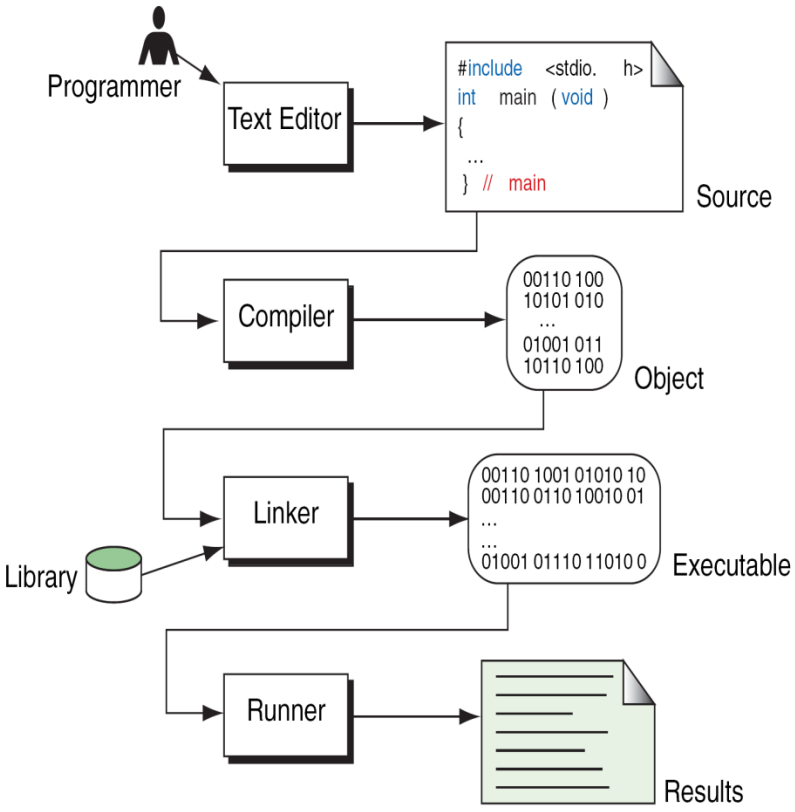
Compiling C program in C Programming Language: Computer does not understand c programming language. It understands only 0 and 1 means machine language. So c programming language code is converted into machine language. The process of converting source code in to machine code is called compiling. Compiler is a program that compiles source code. Compiler also detects errors in source program. If compiling is successful source program is converted into object program. Object program is saved on disk. The extension of file is ".obj"

Linking in C programming Language: There are many built in functions available in c programming language. These functions are also called library functions. These functions are stored in different header files.

Loading program: The process of transferring a program from secondary storage to main memory for execution is called loading a program. A program called loader

does loading.

Executing program: Execution is the last step. In this step program starts execution. Its instructions start working and output of the program display on the screen.



Pseudocode: is an artificial and informal language that helps programmers develop algorithms. Pseudocode is very similar to everyday English.

Algorithm:

An algorithm is a description of a procedure which terminates with a result. Algorithm is a step-by-step method of solving a problem.

Properties of an Algorithm:
1) Finiteness: - An algorithm terminates after a finite numbers of steps.
2) Definiteness: - Each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.
3) Input: - An algorithm accepts zero or more inputs
4) Output:- An algorithm should produce at least one output.

5) Effectiveness: - It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

Writing an algorithm

An algorithm can be written in English, like sentences and using mathematical formulas. Sometimes algorithm written in English like language is Pseudo code.

Examples

1) Finding the average of three numbers

1. Let a,b,c are three integers
2. Let d is float
3. Display the message "Enter any three integers:"
4. Read three integers and stores in a,b,c
5. Compute the d = (a+b+c)/3.0
6. Display "The avg is:" , d
7. End.

- **Example 1:** Write an algorithm to determine a student's final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.

Pseudocode::

- **Input a set of 4 marks**
- **Calculate their average by summing and dividing by 4**
- **if average is below 50**

```
        Print "FAIL"
else
        Print "PASS"
```

- **Detailed Algorithm :**
-

```
Step 1:   Input M1,M2,M3,M4
Step 2:   GRADE ← (M1+M2+M3+M4)/4
Step 3:   if (GRADE < 50) then
                Print "FAIL"
          else
                Print "PASS"
          endif
```

Flowcharts :

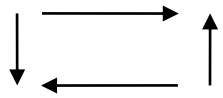
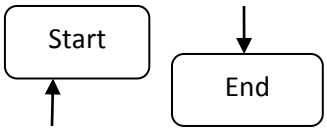
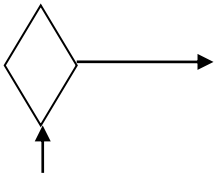
The pictorial representation of algorithm is called flowchart.

Uses of flow chart:

1 : flow chart helps to understand the program easily.

2 : as different symbols are used to specify the type of operation performed, it is easier to understand the complex programs with the help of flowcharts.

Flowchart Symbols

S.NO	Description	Symbols
1	**Flowlines :** These are the left to right or top to bottom lines connection symbols. These lines shows the flow of control through the program.	
2	**Terminal Symbol :** The oval shaped symbol always begins and ends the flowchart. Every flow chart starting and ending symbol is terminal symbol.	
3	**Input / Output symbol :** The parallelogram is used for both input (Read) and Output (Write) is called I/O symbol. This symbol is used to denote any function of an I/O device in the program.	
4	**Process Symbol :** The rectangle symbol is called process symbol. It is used for calculations and initialization of memory locations.	
5	**Decision symbol :** The diamond shaped symbol is called decision symbol. This box is used for decision making. There will be always two exists from a decision symbol one is labeled YES and other labeled NO.	
6	**Connectors :** The connector symbol is represented by a circle. Whenever a complex flowchart is morethan one page, in such a situation, the connector symbols are used to connect the flowchart.	

Algorithm to find whether a number even or odd:

Step1: Begin
Step2: Take a number
Step3: if the number is divisible by2 then
print that number is even
otherwise print that number is odd

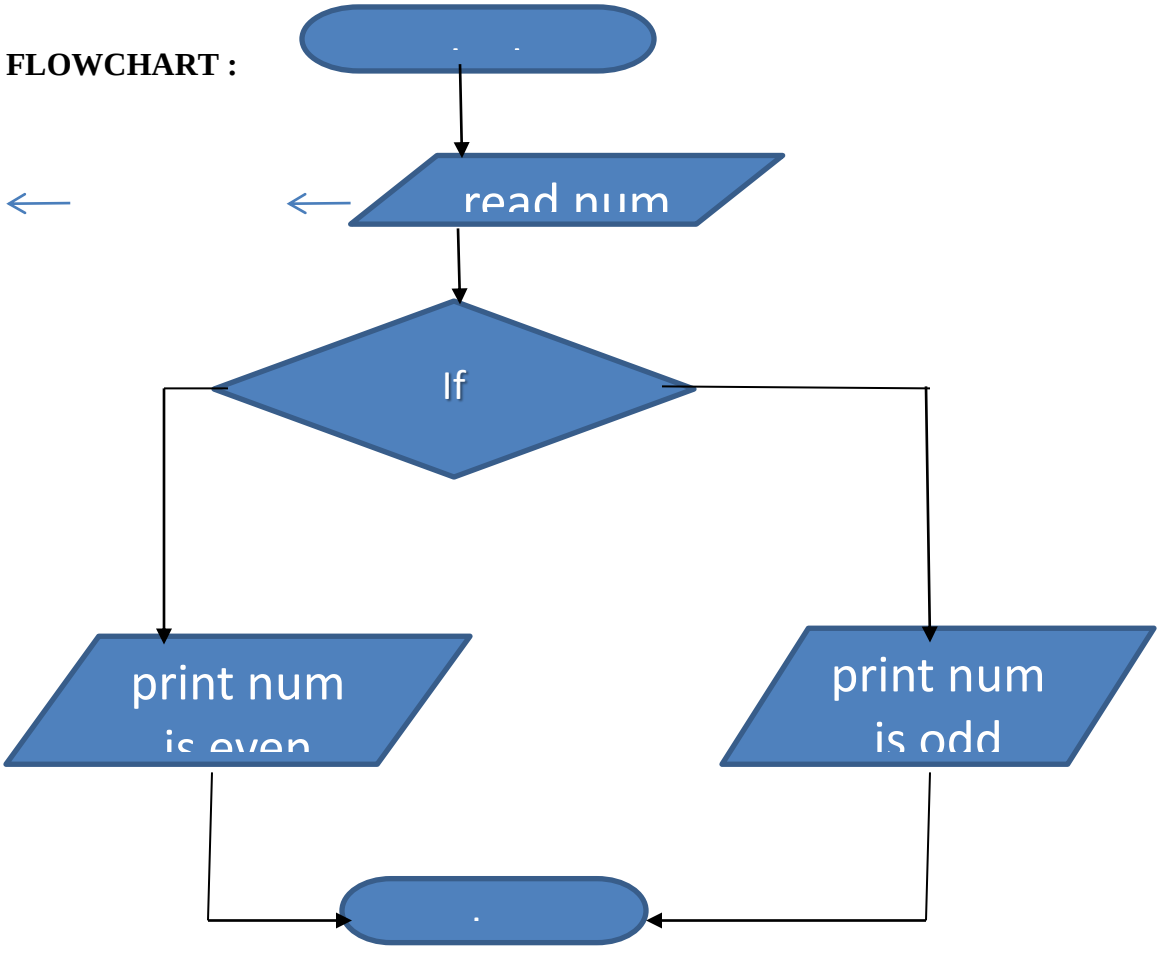
Step1: START
Step2: Read num
Step3: if(num%2=0) then
print num is even
otherwise

print num is odd

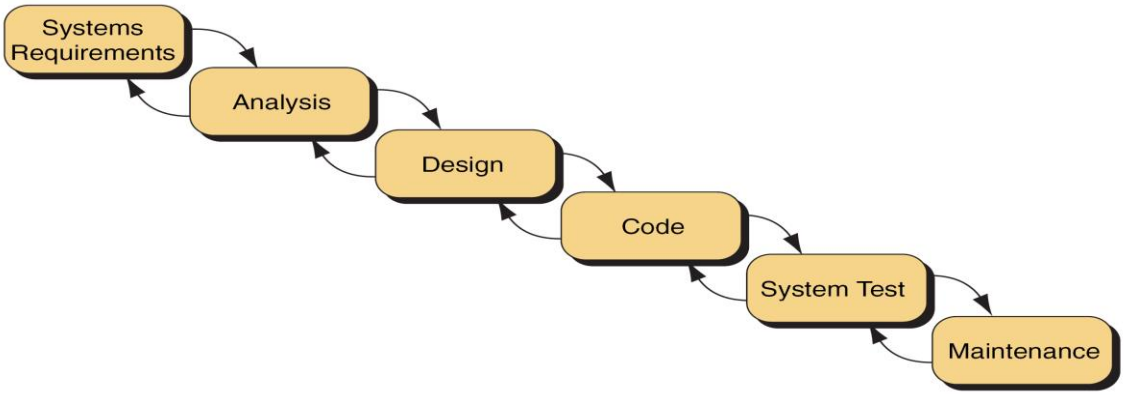
Step4: End
(Algorithm in natural language)

Step4: STOP
(Algorithm by using pseudo code)

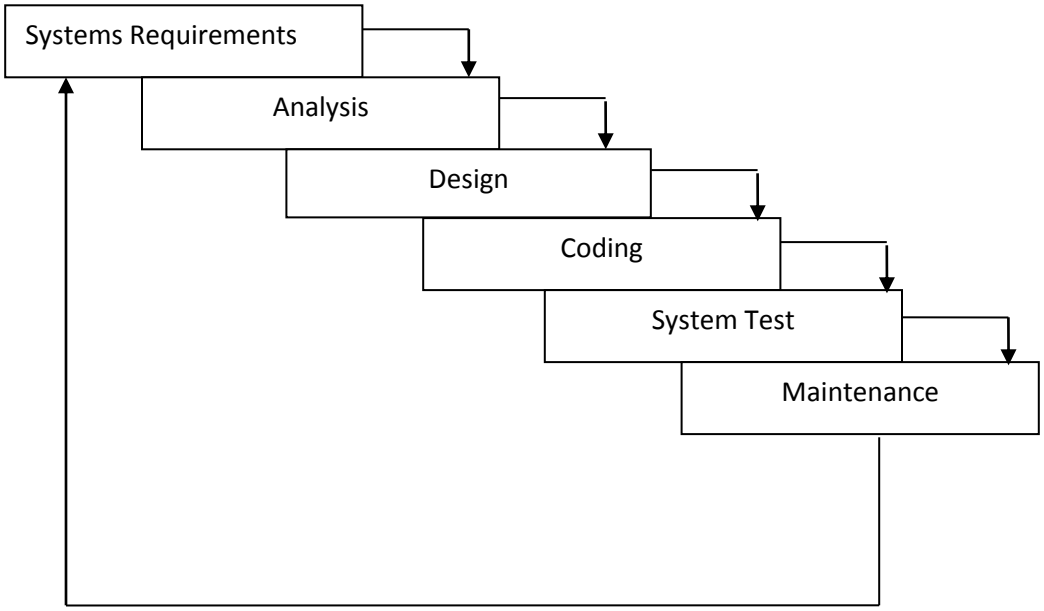
FLOWCHART :



System Development:



Or



1. Statement of Problem

a) Working with existing system and using proper questionnaire, the problem should be explained
clearly.
b) What inputs are available, what outputs are required and what is needed for creating workable solution, should be understood clearly.

2. Analysis
a) The method of solutions to solve the problem can be identified.
b) We also judge that which method gives best results among different methods of solution.

3. Design
a) Algorithms and flow charts will be prepared.
b) Focus on data, architecture, user interfaces and program components.

4. System Test
The algorithms and flow charts developed in the previous steps are converted into actual programs in the high level languages like C.

a. Compilation
The process of translating the program into machine code is called as Compilation. Syntactic errors are found quickly at the time of compiling the program. These errors occur due to the usage of wrong syntaxes for the statements.
Eg: x=a*y+b
There is a syntax error in this statement, since, each and every statement in C language ends with a semicolon (;).

b. Execution
The next step is Program execution. In this phase, we may encounter two types of errors. Runtime Errors: these errors occur during the execution of the program and terminate the program abnormally.
Logical Errors: these errors occur due to incorrect usage of the instructions in the program. These errors are neither detected during compilation or execution nor cause any stoppage to the program execution but produces incorrect output.

5. Maintenance
We are maintenance the software by updating the information, providing the security and license for the software.

What is C?

C is a programming language developed at **AT & T's Bell Laboratories of USA in 1972**. It was designed and written by **Dennis Ritche**. **Dennis Ritchie** is known as the **founder of c language**.

It was developed to overcome the problems of previous languages such as B, BCPL etc.

Initially, C language was developed to be used in UNIX operating system.

Features of C

1. Portability or machine independent

2. Sound and versatile language
3. Fast program execution.
4. An extendible language.
5. Tends to be a structured language.

Historical developments of C(Background)

Year	Language	Developed by	Remarks
1960	ALGOL	International committee	Too general, too abstract
1967	BCPL	Martin Richards at Cambridge university	Could deal with only specific problems
1970	B	Ken Thompson at AT & T	Could deal with only specific problems
1972	C	Dennis Ritche at AT & T	Lost generality of BCPL and B restored

General Structure of a C program:

```
/* Documentation section */
/* Link section */
/* Definition section */
/* Global declaration section */
main()
{
Declaration part
Executable part (statements)
}
/* Sub-program section */
```

- The documentation section is used for displaying any information about the program like the purpose of the program, name of the author, date and time written etc, and this section should be enclosed within comment lines. The statements in the documentation section are ignored by the compiler.
- The link section consists of the inclusion of header files.

- The definition section consists of macro definitions, defining constants etc,.
- Anything declared in the global declaration section is accessible throughout the program, i.e. accessible to all the functions in the program.
- main() function is mandatory for any program and it includes two parts, the declaration part and the executable part.
- The last section, i.e. sub-program section is optional and used when we require including user defined functions in the program.

First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main(){
4. printf("Hello C Language");
5. getch();
6. }
```

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .

#include <conio.h> includes the **console input output** library functions. The getch() function is defined in conio.h file.

void main() The **main() function is the entry point of every program** in c language. The void keyword specifies that it returns no value.

printf() The printf() function is **used to print data** on the console.

getch() The getch() function **asks for a single character**. Until you press any key, it blocks the screen.

C TOKENS: The smallest individual units are known as tokens. C has six types of tokens.

1: Identifiers

2: Keywords

3: Constants

4: Strings

5: Special Symbols

6: Operators

Identifiers:

Identifiers refer to the names of variables, constants, functions and arrays. These are user-defined names is called Identifiers. These identifier are defined against a set of rules.

Rules for an Identifier

1. An Identifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
2. The first character of an identifier can only contain alphabet(a-z , A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example *name* and *Name* are two different identifier in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.
6. C' compiler recognizes only the first 31 characters of an identifiers.

Ex :	**Valid**	**Invalid**
	STDNAME	Return
	SUB	*$stay*
	TOT_MARKS	1RECORD
	_TEMP	STD NAME.
	Y2K	

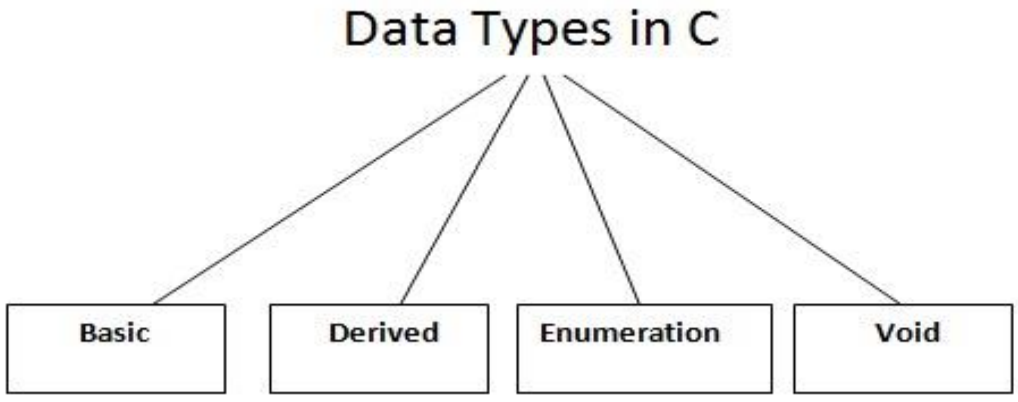
Keywords: A keyword is a **reserved word.** All keywords have fixed meaning that means we cannot change. Keywords serve as basic building blocks for program statements. All keywords must be written in lowercase. A list of 32 keywords in c language is given below:

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

Note: Keywords we cannot use it as a variable name, constant name etc.

Data Types/Types:

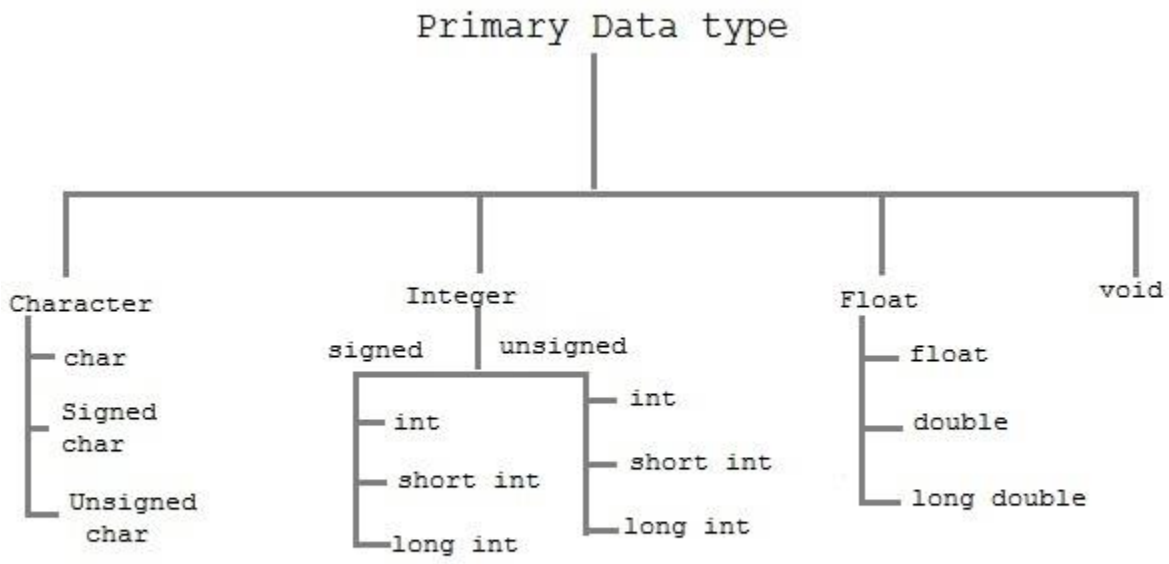
- To store data the program must reserve space which is done using datatype. A datatype is a keyword/predefined instruction used for allocating memory for data. A data type specifies the type of data that a variable can store such as integer, floating, character etc. It used for declaring/defining variables or functions of different types before to use in a program.



There are 4 types of data types in C language.

Types	Data Types
Basic Data Type	**int, char, float, double**
Derived Data Type	**array, pointer, structure, union**
Enumeration Data Type	**enum**
Void Data Type	**void**

Note: We call Basic or Primary data type.



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

Size and Ranges of Data Types with Type Qualifiers

Type	Size (bytes)	Range	Control String
char or signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c

int or signed int	2	-32768 to 32767	%d or %i
unsigned int	2	0 to 65535	%u
short int or signed short int	1	-128 to 127	%d or %i
unsigned short int	1	0 to 255	%d or %i
long int or signed long int	4	-2147483648 to 2147483647	%ld
unsigned long int	4	0 to 4294967295	%lu
float	4	3.4E-38 to 3.4E+38	%f or %g
double	8	1.7E-308 to 1.7E+308	%lf
long double	10	3.4E-4932 to 1.1E+4932	%Lf

Variables

A **variable** is a name of memory location. It is used to store data. Variables are changeable, we can change value of a variable during execution of a program. . It can be reused many times.

Note: Variable are nothing but identifiers.

Rules to write variable names:

1. A variable name contains maximum of 30 characters/ Variable name must be upto 8 characters.
2. A variable name includes alphabets and numbers, but it must start with an alphabet.
3. It cannot accept any special characters, blank spaces except under score(_).
4. It should not be a reserved word.

Ex : i rank1 MAX min Student_name StudentName class_mark

Declaration of Variables : A variable can be used to store a value of any data type. The declaration of variables must be done before they are used in the program. The general format for declaring a variable.

Syntax : data_type variable-1,variable-2,------, variable-n;
Variables are separated by commas and declaration statement ends with a semicolon.

Ex : int x,y,z;
float a,b;
char m,n;

Assigning values to variables : values can be assigned to variables using the assignment operator (=). The general format statement is :

Syntax : variable = constant;

Ex : x=100;
a= 12.25;
m='f';

we can also assign a value to a variable at the time of the variable is declared. The general format of declaring and assigning value to a variable is :

Syntax : data_type variable = constant;

Ex ; int x=100;
float a=12.25;
char m='f';

Types of Variables in C

There are many types of variables in c:

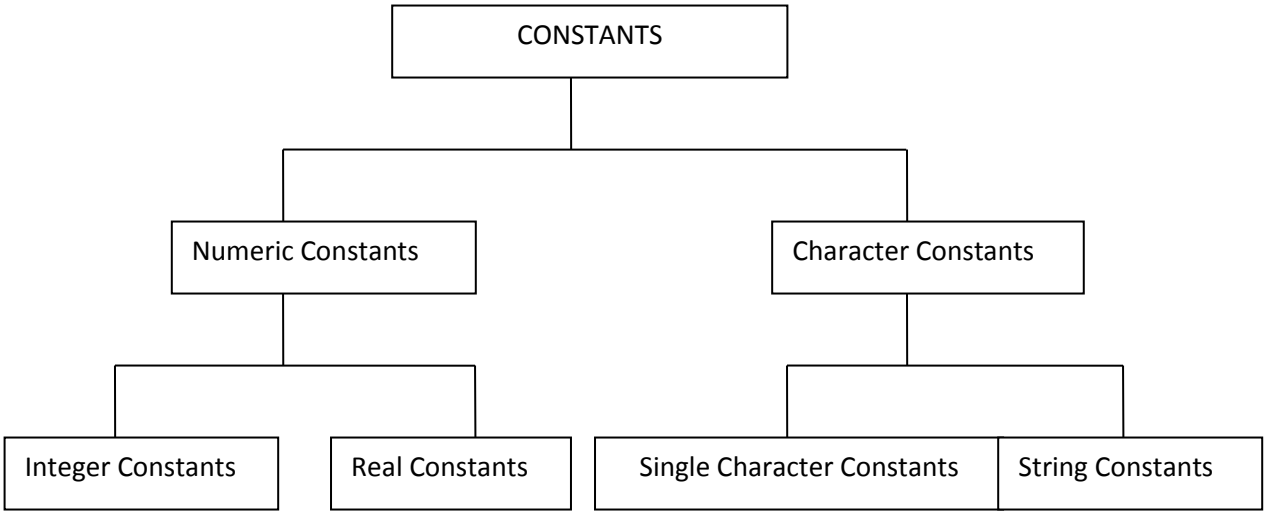
1. local variable
2. global variable
3. static variable

Constants

Constants **refer** to fixed values that do not change during the execution of a program.

Note: constants are also called literals.

C supports several kinds of constants.



TYPES OF C CONSTANT:

1. Integer constants
2. Real or Floating point constants
3. Character constants
4. String constants
5. Backslash character constants

Integer constants:

An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

- decimal constant(base 10)
- octal constant(base 8)
- hexadecimal constant(base 16)

For example:

- Decimal constants: 0, -9, 22 etc
- Octal constants: 021, 077, 033 etc
- Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

- In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

1: Decimal Integer : the rules for represent decimal integer.

a) Decimal Integer value which consist of digits from 0-9.
b) Decimal Integer value with base 10.
c) Decimal Integer should not prefix with 0.
d) It allows only sign (+,-).
e) No special character allowed in this integer.

Ex :	**valid**	**invalid**
	7	$77
	77	077
	+77	7,777
	-77	

2 : Octal : An integer constants with base 8 is called octal. These rules are :

a) it consist of digits from 0 to 7.
b) It should prefix with 0.
c) It allows sign (+,-).
d) No special character is allowed.

EX :	**VALID**	**INVALID**
	0123	123 -> it because no prefix with 0
	+0123	0128 -> because digits from 0 to 7.
	-0123	

3 : Hexadecimal : An integer constant with base value 16 is called Hexadecimal.

a) It consist of digits from 0-9,a-f(capital letters & small leters.

Ex : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

b) it should prefix with 0X or 0x.
c) it allows sign (+,-).
d) No special character is allowed.

EX : OX1a, ox2f

Floating point/Real constants:

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

Note: E-5 = 10^{-5}

Real Constants : Real constant is base 10 number, which is represented in decimal 0r scientific/exponential notation.

Real Notation : The real notation is represented by an integer followed by a decimal point and the fractional(decimal) part. It is possible to omit digits before or after the decimal point.

Ex : 15.25
.75
30
-9.52
-92
+.94

***Scientific/Exponential Notation*:** The general form of Scientific/Exponential notation is

mantisha e exponent

The **mantisha** is either a real/floating point number expressed in decimal notation or an integer and the **exponent** is an integer number with an optional sign. The character **e** separating the mantisha and the exponent can be written in either lowercase or uppercase.

Ex : 1.5E-2
100e+3
-2.05e2

Character Constant:

Single Character Constant : A character constant is either a single alphabet, a single digit, a single special symbol enclosed within single inverted commas.

a) it is value represent in ‘ ‘ (single quote).
b) The maximam length of a character constant can be 1 character.

EX : VALID INVALID
‘a’ “12”

'A' 'ab'

String constant : **A** string constant is a sequence of characters enclosed in double quote, the characters may be letters, numbers, special characters and blank space etc

EX : "rama" , "a" , "+123" , "1-/a"

"good" //string constant

"" //null string constant

" " //string constant of six white space

"x" //string constant having single character.

"Earth is round\n" //prints string with newline

Escape characters or backslash characters:

a) \n newline
b) \r carriage return
c) \t tab
d) \v vertical tab
e) \b backspace
f) \f form feed (page feed)
g) \a alert (beep)
h) \' single quote(')
i) \" double quote(")
j) \? Question mark (?)
k) \\ backslash (\)

Two ways to define constant in C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor
3.

1) C const keyword

The const keyword is used to define constant in C programming.

1. **const float** PI=3.14;

Now, the value of PI variable can't be changed.

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main(){
4. const float PI=3.14;
5. clrscr();
6. printf("The value of PI is: %f",PI);
```

```
7. getch();
8. }
```
Output:
The value of PI is: 3.140000

2) C #define preprocessor

The #define preprocessor is also used to define constant.

C#define

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

#define token value

Let's see an example of #define to define a constant.

```
#include <stdio.h>
1. #define PI 3.14
2. main() {
3.     printf("%f",PI);
4. }
```
Output:
3.140000

Formatted and Unformatted Console I/O Functions.

Input / Output (I/O) Functions : In 'C' language, two types of Input/Output functions are available, and all input and output operations are carried out through function calls. Several functions are available for input / output operations in 'C'. These functions are collectively known as the standard i/o library.

Input: In any programming language input means to feed some data into program. This can be given in the form of file or from command line.

Output: In any programming language output means to display some data on screen, printer or in any file.

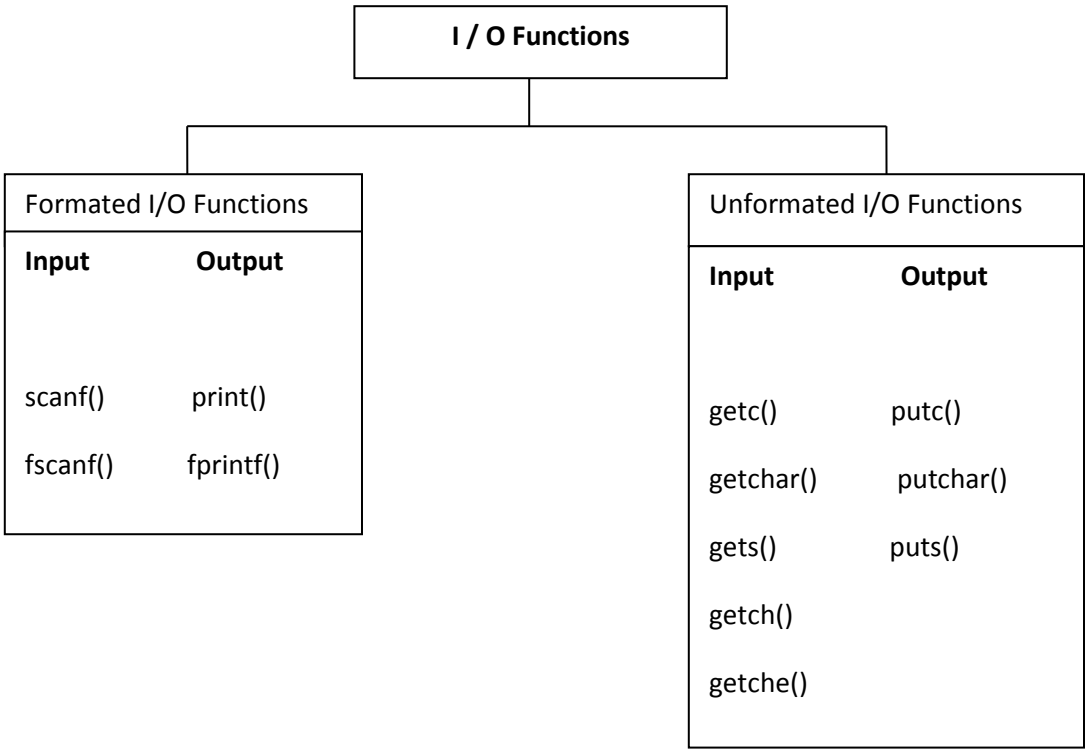
The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard

Standard output	stdout	Screen
Standard error	stderr	Your screen

Input / Output functions are classified into two types



. **Formated I/O Functions :** formatted I/O functions operates on various types of data.

1 : printf() : output data or result of an operation can be displayed from the computer to a standard output device using the library function printf(). This function is used to print any combination of data.

Syntax : printf("control string ", variable1, variable2, ----------, variablen);

Ex : printf("%d",3977); // **Output**: 3977

printf() statement another syntax :

Syntax : **printf(**"fomating string");

Formating string : it prints all the character given in doublequotes (" ") except formatting specifier.

Ex : printf(“ hello “);-> hello
printf(“a”); -> a
printf(“%d”, a); -> a value
printf(“%d”); -> no display

scanf() : input data can be entered into the computer using the standard input ‘C’ library function called scanf(). This function is used to enter any combination of input.

Syntax : scanf(“control string “,&var1, &var2,----, &varn);

The scanf() function is used to read information from the standard input device (keyboard).

Ex : scanf(“ %d “,&a);-> hello

Each variable name (argument) must be preceeded by an ampersand (&). The (&) symbol gives the meaning “address of “ the variable.

Unformatted I/O functions:

a) **Character I/O**
b) **String I/O**

a) character I/O:

1. getchar(): Used to read a character from the standard input
2. putchar(): Used to display a character to standard output
3. getch() and getche(): these are used to take the any alpha numeric characters from the standard input
   getche() read and display the character
   getch() only read the single character but not display
4. putch(): Used to display any alpha numeric characters to standard output

a) String I/O:

1. gets(): Used for accepting any string from the standard input(stdin) eg:gets()

2. puts(): Used to display a string or character array Eg:puts()
3. Cgets():read a string from the console eg; cgets(char *st)
4. Cputs():display the string to the console eg; cputs(char *st)

OPERATORS AND EXPRESSIONS:

Operators : An operator is a Symbol that performs an operation. An operators acts some variables are called operands to get the desired result.

Ex : a+b;

Where a,b are operands and + is the operator.

Types of Operator :

1) Arithmetic Operators.
2) Relational Operators.
3) Logical Operators.
4) Assignment Operators.
5). Unary Operators.
6) Conditional Operators.
7) Special Operators.
8) Bitwise Operators.
9) Shift Operators.

Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

C Program to demonstrate the working of arithmetic operators

```
#include <stdio.h>
void  main()
{
   int a = 9,b = 4, c;

   c = a+b;
   printf("a+b = %d \n",c);

   c = a-b;
   printf("a-b = %d \n",c);

   c = a*b;
   printf("a*b = %d \n",c);

   c=a/b;
   printf("a/b = %d \n",c);

   c=a%b;
   printf("Remainder when a divided by b = %d \n",c);


}
```

Output
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1

Relational Operators. A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.
Operands may be variables, constants or expressions.

Relational operators are used in decision making and loops.

Operator	Meaning	Example	Return value
<	is less than	2<9	1
< =	is less than or equal to	2 < = 2	1
>	is greater than	2 > 9	0
> =	is greater than or equal to	3 > = 2	1
= =	is equal to	2 = = 3	0
!=	is not equal to	2!=2	0

```
// C Program to demonstrate the working of relational operators

#include <stdio.h>

int main()

{

    int a = 5, b = 5, c = 10;

    printf("%d == %d = %d \n", a, b, a == b); // true

    printf("%d == %d = %d \n", a, c, a == c); // false

    printf("%d > %d = %d \n", a, b, a > b); //false

    printf("%d > %d = %d \n", a, c, a > c); //false
```

```
    printf("%d < %d = %d \n", a, b, a < b); //false
    printf("%d < %d = %d \n", a, c, a < c); //true
    printf("%d != %d = %d \n", a, b, a != b); //false
    printf("%d != %d = %d \n", a, c, a != c); //true
    printf("%d >= %d = %d \n", a, b, a >= b); //true
    printf("%d >= %d = %d \n", a, c, a >= c); //false
    printf("%d <= %d = %d \n", a, b, a <= b); //true
    printf("%d <= %d = %d \n", a, c, a <= c); //true
    return 0;
}
```

Output

```
5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
```

5 <= 5 = 1

5 <= 10 = 1

Logical Operators.

These operators are used to combine the results of two or more conditions. An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example	Return value
&&	Logical AND	(9>2)&&(17>2)	1
||	Logical OR	(9>2) || (17 = = 7)	1
!	Logical NOT	29!=29	0

Logical AND : If any one condition false the complete condition becomes false.

Truth Table

Op1	Op2	Op1 && Op2
true	true	true
true	false	false
false	true	false
false	false	false

Logical OR : If any one condition true the complete condition becomes true.

Truth Table

Op1	Op2	Op1 // Op2
true	true	true
true	false	true
false	true	true
false	false	false

Logical Not : This operator reverses the value of the expression it operates on i.e, it makes a true expression false and false expression true.

Op1	Op1 !
true	false
false	true

// C Program to demonstrate the working of logical operators

#include <stdio.h>

```
int main()
{
   int a = 5, b = 5, c = 10, result;
   result = (a = b) && (c > b);
   printf("(a = b) && (c > b) equals to %d \n", result);
   result = (a = b) && (c < b);
   printf("(a = b) && (c < b) equals to %d \n", result);
   result = (a = b) || (c < b);
   printf("(a = b) || (c < b) equals to %d \n", result);
   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) equals to %d \n", result);
   result = !(a != b);
   printf("!(a == b) equals to %d \n", result);
   result = !(a == b);
   printf("!(a == b) equals to %d \n", result);
   return 0;
}
```

Output

```
(a = b) && (c > b) equals to 1
(a = b) && (c < b) equals to 0
(a = b) || (c < b) equals to 1
```

(a != b) || (c < b) equals to 0

!(a != b) equals to 1

!(a == b) equals to 0

Assignment Operators. Assignment operators are used to assign a value (or) an expression (or) a value of a variable to another variable.

Syntax : variable name=expression (or) value (or) variable

Ex : x=10;
y=a+b;
z=p;

Compound assignment operator:

'C' provides compound assignment operators to assign a value to variable in order to assign a new value to a variable after performing a specified operation.

Operator	Example	Meaning
+ =	x + = y	x=x+y
- =	x - = y	x=x-y
* =	x * = y	x=x*y
/ =	x / = y	x=x/y
% =	x % = y	X=x%y

// C Program to demonstrate the working of assignment operators

```
#include <stdio.h>

int main()

{

   int a = 5, c;

   c = a;
```

```
    printf("c = %d \n", c);
    c += a; // c = c+a
    printf("c = %d \n", c);
    c -= a; // c = c-a
    printf("c = %d \n", c);
    c *= a; // c = c*a
    printf("c = %d \n", c);
    c /= a; // c = c/a
    printf("c = %d \n", c);
    c %= a; // c = c%a
    printf("c = %d \n", c);
    return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

Increment and Decrement Operators /Unary Operators:

Unary operators are having higher priority than the other operators. **Unary operators**, meaning **they only operate on a single operand.**

Increment Operator in C Programming

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 ++.

We have two types of increment operator i.e Pre-Increment and Post-Increment Operator**.**

Pre-Increment

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

b = ++y;

In this example suppose the value of variable 'y' is 5 then value of variable 'b' will be 6 because the value of 'y' gets modified before using it in a expression**.**

Post-Increment

Post-increment operator is used to increment the value of variable as soon as after executing expression completely in which post increment is used. In the Post-Increment value is first used in a expression and then incremented.

b = x++;

In this example suppose the value of variable 'x' is 5 then value of variable 'b' will be 5 because old value of 'x' is used.

Note :

We cannot use increment operator on the constant values because increment operator operates on only variables. It increments the value of the variable by 1 and stores the incremented value back to the variable

b = ++5;

or

b = 5++;

The **syntax** of the operators is given below.

++<variable name> --<variable name>
<variable name>++ <variable name>--

The operator ++ adds 1 to the operand and – subtracts 1 from the operand. These operators in two forms : prefix (++x) and postfix(x++).

Operator	**Meaning**
++x	Pre increment
- -x	Pre decrement
x++	Post increment
x--	Post decrement

Where
1 : ++x : Pre increment, first increment and then do the operation.
2 : - -x : Pre decrement, first decrements and then do the operation.
3 : x++ : Post increment, first do the operation and then increment.
4 : x- - : Post decrement, first do the operation and then decrement.

```
// C Program to demonstrate the working of increment and decrement operators
#include <stdio.h>
int main()
{
   int a = 10, b = 100;
   float c = 10.5, d = 100.5;
   printf("++a = %d \n", ++a);
   printf("--b = %d \n", --b);
   printf("++c = %f \n", ++c);
   printf("--d = %f \n", --d);
   return 0;
}
```

Output

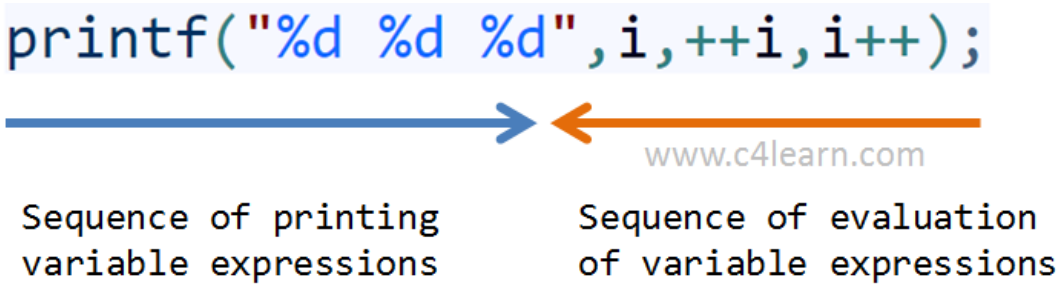
```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Multiple increment operators inside printf

```
#include<stdio.h>
void main() {
  int i = 1;
  printf("%d %d %d", i, ++i, i++);
}
```

Output : 3 3 1

Pictorial representation



Explanation of program

I am sure you will get confused after viewing the above image and output of program.

1. Whenever more than one format specifiers (i.e %d) are directly or indirectly related with same variable (i,i++,++i) then we need to evaluate each individual expression from right to left.
2. As shown in the above image evaluation sequence of expressions written inside printf will be – i++,++i,i
3. After execution we need to replace the output of expression at appropriate place

No	Step	Explanation
1	Evaluate i++	At the time of execution we will be using older value of i = 1
2	Evaluate ++i	At the time of execution we will be increment value already modified after step 1 i.e i = 3
2	Evaluate i	At the time of execution we will be using value of i modified in step 2

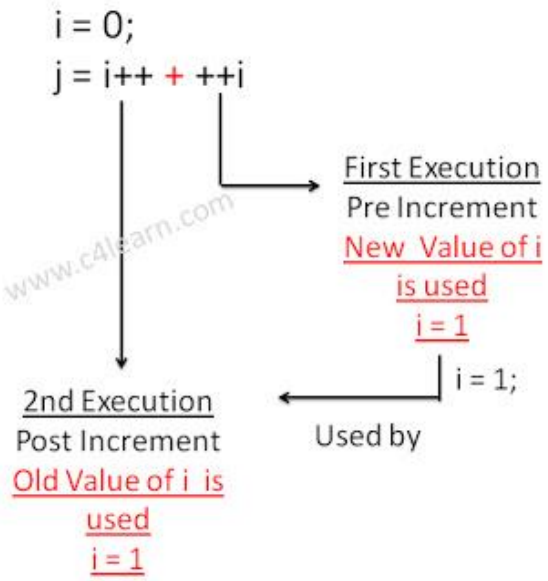
Postfix and Prefix Expression in Same Statement

```
#include<stdio.h>
#include<conio.h>
void main() {
   int i = 0, j = 0;
   j = i++ + ++i;
   printf("%d\n", i);
   printf("%d\n", j);
}
```

Output :

2
2

Explanation of Program



Conditional Operator/ Ternary operator:

conditional operator checks the condition and executes the statement depending of the condition.
A conditional operator is a ternary operator, that is, it works on 3 operands.
Conditional operator consist of two symbols.

1 : question mark (?).
2 : colon (:).

Syntax : condition ? exp1 : exp2;

It first evaluate the condition, if it is true (non-zero) then the "exp1" is evaluated, if the condition is false (zero) then the "exp2" is evaluated.

```
#include <stdio.h>
int main(){
  char February;
  int days;
  printf("If this year is leap year, enter 1. If not enter any integer: ");
  scanf("%c",&February);
  // If test condition (February == 'l') is true, days equal to 29.
  // If test condition (February =='l') is false, days equal to 28.
  days = (February == '1') ? 29 : 28;
  printf("Number of days in February = %d",days);
  return 0;
}
```

Output

If this year is leap year, enter 1. If not enter any integer: 1
Number of days in February = 29

Bitwise Operators:

Bitwise operators are used to manipulate the data at bit level. **It operates on integers only. It may not be applied to float.**In arithmetic-logic unit (which is within the CPU), mathematical operations like: addition, subtraction, multiplication and division are done in bit-level which makes processing faster and saves power. To perform bit-level operations in C programming, bitwise operators are used.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
< <	Shift left
> >	Shift right
~	One's complement.

Bitwise AND operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

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)

Example #1: Bitwise AND

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

Output

Output =8

Bitwise OR operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 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)

Example #2: Bitwise OR

```
#include <stdio.h>

int main()

{

   int a = 12, b = 25;

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

   return 0;
```

```
}
```

Output

Output =29

Bitwise XOR (exclusive OR) operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. 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)
```

Example #3: Bitwise XOR

```
#include <stdio.h>

int main()

{

   int a = 12, b = 25;

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

   return 0;

}
```

Output

Output = 21

Bitwise complement operator ~

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. 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

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n+1). To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that 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.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

Bitwise complement of any number N is -(N+1). Here's how:

bitwise complement of N = ~N (represented in 2's complement form)

2'complement of ~N= -(~(~N)+1) = -(N+1)

Example #4: Bitwise complement

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

```
int main()

{

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

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

   return 0;

}
```

Output

Complement = -36

Complement = 11

There are two Bitwise shift operators in C programming:

- Right shift operator
- Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

Special Operators

1) Comma Operator :The comma operator is used to separate the statement elements such as variables, constants or expressions, and this operator is used to link the related expressions together, such expressions can be evaluated from left to right and the value of right most expressions is the value of combined expressions

Ex : val(a=3, b=9, c=77, a+c)

First signs the value 3 to a, then assigns 9 to b, then assigns 77 to c, and finaly80(3+77) to value.

2) Sizeof Operator : The sizeof() is a unary operator, that returns the length in bytes o the specified variable, and it is very useful to find the bytes occupied by the specified variable in the memory.

Syntax : **sizeof(variable-name);**

```
        int a;
Ex : sizeof(a);      //OUTPUT-----2bytes
```

Example #6: sizeof Operator

```
#include <stdio.h>
int main()
{
   int a, e[10];
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));
   printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
   return 0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
Size of integer type array having 10 elements = 40 bytes
```

Expressions

Expressions : An expression is a combination of operators and operands which reduces to a single value. An operator indicats an operation to be performed on data that yields a value. An operand is a data item on which an operation is performed.

A simple expression contains only one operator.

Ex : 3+5 is a simple expression which yields a value 8, -a is also a single expression.
A complex expression contain more than one operator.

Ex : complex expression is 6+8*7.

Ex ;	**Algeberic Expressions**	**C-expression**
	1 : ax2+bx+c	1: a*x*x+b*x+c
	2 : a+bx	2 : a+b*x.
	3 : 4ac/b	3 : 4*a*c/b.
	4 : x2/y2-1	4 : x*x/y*y-1

Operator Precedence : Arithmetic Operators are evaluvated left to right using the precedence of operator when the expression is written without the paranthesis.They are two levels of arithmetic operators in C.

1 : High Priority * / %
2 : Low Priority + -.

Arithmetic Expression evaluation is carried out using the two phases from left to right.

1 : First phase : The highest priority operator are evaluated in the 1st phase.
2 : Second Phase : The lowest priority operator are evaluated in the 2nd phase.

Ex : a=x-y/3+z*2+p/4.
x=7, y=9, z=11, p=8.
a= 7-9/3+11*2+8/4.

1st phase :
1 : a = 7-3+11*2+8/4
2 : a = 7-3+22+8/4
3 : a = 7-3+22+2
2nd phase :
1 : a = 4+22+2
2 : a = 26+2
3 : a = 28

The order of evaluation can be changed by putting paranthesis in an expression.
Ex : 9-12/(3+3)*(2-1)

Whenever parentheses are used, the expressions within parantheses highest priority. If two or more sets of paranthesis appear one after another. The expression contained in the left-most set is evaluated first and the right-most in the last.
1st phase :
1 : 9-12/6*(2-1)
2 : 9-12/6*1
2nd phase :
1 : 9-2*1
2 : 9-2.
3rd phase :

1 : 7.

Rules for Evaluation of Expression :

1 : Evaluate the sub-expression from left to right. If parenthesized.
2 : Evaluate the arithemetic Expression from left to right using the rules of precedence.
3 : The highest precedence is given to the expression with in paranthesis.
4 : When parantheses are used, the expressions within parantheses assume highest priority.
5 : Apply the associative rule, if more operators of the same precedence occurs.

Operator Precedence and Associativity :

Every operator has a precedence value. An expression containing more than one oerator is known as complex expression. Complex expressions are executed according to precedence of operators.

Associativity specifies the order in which the operators are evaluated with the same precedence in a complex expression. Associativity is of two ways, i.e left to ringht and right to left. Left to right associativity evaluates an expression starting from left and moving towards right. Right to left associativity proceds from right to left.

The precedence and associativity of various operators in C.

Operator	Description	Precedence	Associativity
( ) [ ]	Function call Square brackets.	1	L-R (left to right)
+ - ++ -- ! ~ * & sizeof	Unary plus Unary minus Increment Decrement Not operator Complement Pointer operator Address operator Sizeof operator	2	R-L (right to left)
* / %	Multiplication Division Modulo division	3	L-R (left to right)
+ -	Addition Subtraction	4	L-R (left to right)
<< >>	Left shift Right shift	5	L-R (left to right)

< <= > >=	Relational Operator	6	L-R (left to right)
= = !=	Equality Inequality	7	L-R (left to right)
&	Bitwise AND	8	L-R (left to right)
^	Bitwise XOR	9	L-R (left to right)
|	Bitwise OR	10	L-R (left to right)
&&	Logical AND	11	L-R (left to right)
||	Logical OR	12	L-R (left to right)
?:	Conditional	13	R-L (right to left)
= *= /= %= += -= &= ^= <<= >>=	Assignment operator	14	R-L (right to left)
,	Comma operator	15	L-R (left to right)

Type Conversion/Type casting:

Type conversion is used to convert variable from one data type to another data type, and after type casting complier treats the variable as of new 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 the values from one type to another explicitly using the **cast operator.** Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator.

Syntax:

(type_name) expression;

Without Type Casting:

```
1. int f= 9/4;
2. printf("f : %d\n", f );//Output: 2
```

With Type Casting:

```
1. float f=(float) 9/4;
2. printf("f : %f\n", f );//Output: 2.250000
```

Example:

```
#include <stdio.h>

int main()
```

```
{
   printf( "%c\n", (char)65 );
 getchar();
}
```

or

Type Casting - C Programming

Type casting refers to changing an variable of one data type into another. The compiler will automatically change one type of data into another if it makes sense. For instance, if you assign an integer value to a floating-point variable, the compiler will convert the int to a float. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

Type conversion in c can be classified into the following two types:

1. **Implicit Type Conversion**

When the type conversion is performed automatically by the compiler without programmers intervention, such type of conversion is known as implicit type conversion or type promotion.

```
int x;
for(x=97; x<=122; x++)
{
   printf("%c", x);   /*Implicit casting from int to char thanks to %c*/
}
```

2. **Explicit Type Conversion**

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion. The explicit type conversion is also known as type casting.

Type casting in c is done in the following form:

(data_type)expression;

where, data_type is any valid c data type, and expression may be constant, variable or expression.

For example,

```
int x;

for(x=97; x<=122; x++)

{

   printf("%c", (char)x);   /*Explicit casting from int to char*/

}
```

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

All integer types to be converted to float.

All float types to be converted to double.

All character types to be converted to integer.

Example

Consider the following code:

```
int x=7, y=5 ;

float z;

z=x/y; /*Here the value of z is 1*/
```

If we want to get the exact value of 7/5 then we need explicit casting from int to float:

```
int x=7, y=5;

float z;

z = (float)x/(float)y;   /*Here the value of z is 1.4*/
```

Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int. Consider an example of adding a character with an integer −

```
#include <stdio.h>

main()

 {

   int  i = 17;

   char c = 'c'; /* ascii value is 99 */

   int sum;

   sum = i + c;

   printf("Value of sum : %d\n", sum );

}
```

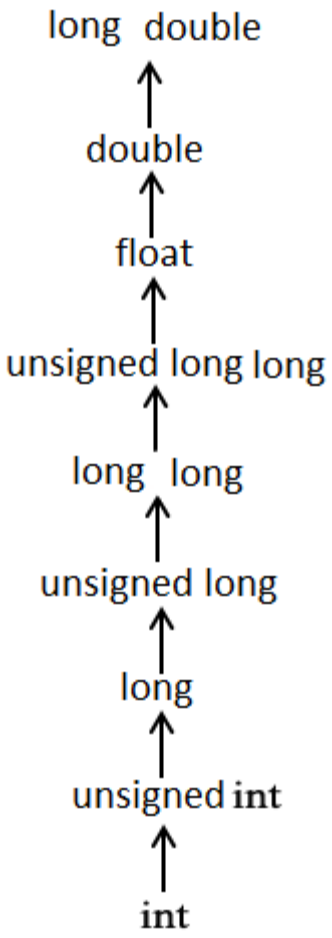
When the above code is compiled and executed, it produces the following result −

```
Value of sum : 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

Usual Arithmetic Conversion

The **usual arithmetic conversions** are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy –



UNIT II

STATEMENTS

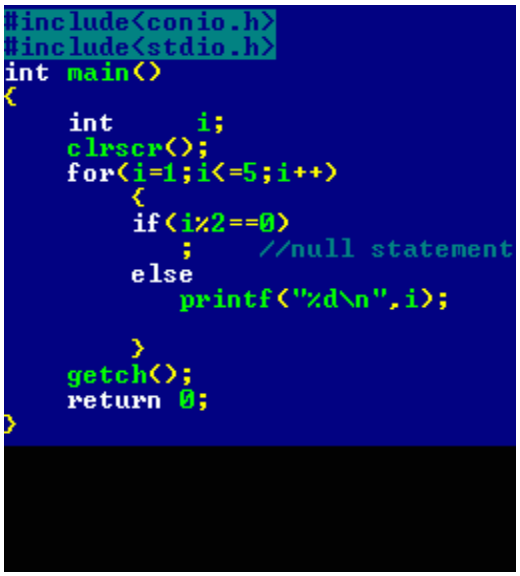
A statement causes the computer to carry out some definite action. There are three different classes of statements in C:

***Expression statements*, C*ompound statements*, and C*ontrol statements*.**

Null statement

A null statement consisting of only a semicolon and performs no operations. It can appear wherever a statement is expected. Nothing happens when a null statement is executed.

Syntax: - ;



```
#include<conio.h>
#include<stdio.h>
int main()
{
    int     i;
    clrscr();
    for(i=1;i<=5;i++)
        {
        if(i%2==0)
            ;    //null statement
        else
            printf("%d\n",i);

        }
    getch();
    return 0;
}
```

Statements such as **do**, **for**, **if**, and while require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

The Null statement is nothing but, there is no body within loop or any other statements in C.

Example illustrates the null statement:

for (i = 0; i < 10; i++) ;

or

for (i=0;i<10;i++)

{

```
//empty body

}
```

Expression

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
i = 0;

i = i + 1;
```

and `printf("Hello, world!\n");`

are all expression statements. In C, however, the semicolon is a statement terminator. Expression statements do all of the real work in a C program. Whenever you need to compute new values for variables, you'll typically use expression statements (and they'll typically contain assignment operators). Whenever you want your program to do something visible, in the real world, you'll typically call a function (as part of an expression statement). We've already seen the most basic example: calling the function printf to print text to the screen.

Note -If no expression is present, the statement is often called the *null statement*.

Return

The return statement terminates execution of a function and returns control to the calling function, with or without a return value. A function may contain any number of return statements. The return statement has

syntax: return *expression*(opt);

If present, the expression is evaluated and its value is returned to the calling function. If necessary, its value is converted to the declared type of the containing function's return value.

A return statement with an expression cannot appear in a function whose return type is void . If there is no expression and the function is not defined as void , the return value is undefined. For example, **the following main function returns an unpredictable value to the operating system:**

```
main ( )

 {
```

return;

}

Compound statements

A compound statement (also called a "block") typically appears as the body of another statement, such as the if statement, for statement, while statement, etc

A Compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression statements, compound statements or control statements. Unlike expression statements, a compound statement does not end with a semicolon. A typical Compound statement is given below.

{

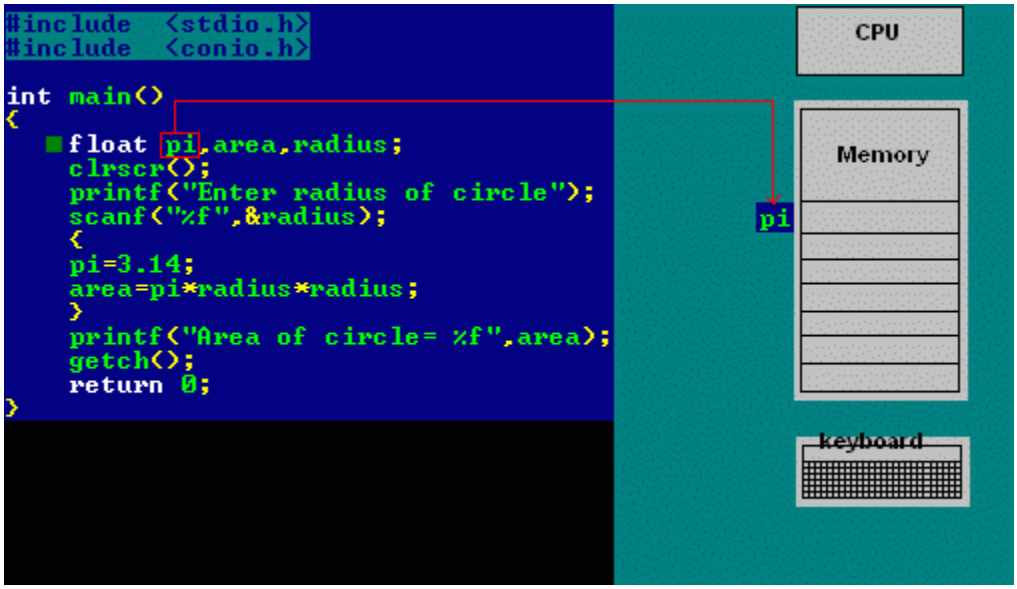
pi=3.14;

area=pi*radius*radius;

}

The particular compound statement consists of two assignment-type expression statements.

Example:



Selection Statement/Conditional Statements/Decision Making Statements

A selection statement selects among a set of statements depending on the value of a controlling expression. Or

Moving execution control from one place/line to another line based on condition

Or

Conditional statements control the sequence of statement execution, depending on the value of a integer expression

C' language supports two conditional statements.

1: if

2: switch.

1: **if Statement:** The if Statement may be implemented in different forms.

1: simple if statement.

2: if –else statement

3: nested if-else statement.

4: else if ladder.

if statement.

The if statement controls conditional branching. The body of an if statement is executed if the value of the **expression is nonzero. Or if statement** is used to execute the code **if condition is true.** If the expression/condition is evaluated to false (0), statements inside the body of if is skipped from execution.

Syntax : if(condition/expression)

{

true statement;

}

statement-x;

If the condition/expression is true, then the true statement will be executed otherwise the true statement block will be skipped and the execution will jump to the statement-x. The 'true statement' may be a single statement or group of statement.

If there is only one statement in the if block, then the braces are optional. But if there is more than one statement the braces are compulsory

Flowchart

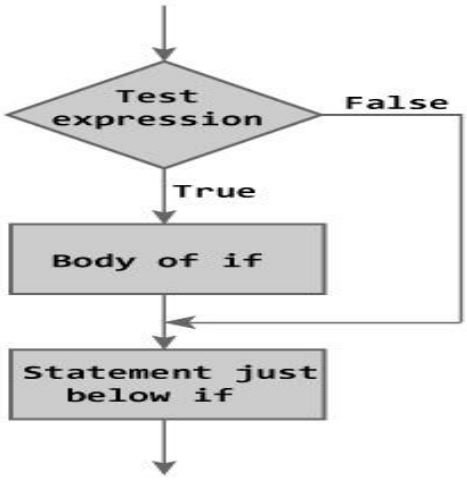


Figure: Flowchart of if Statement

Example:

```
#include<stdio.h>

main()

{

 int a=15,b=20;
```

```
 if(b>a)

{

printf("b is greater");

}

}
```

Output

```
b is greater
```

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

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

   // Test expression is true if number is less than 0
   if (number < 0)
   {
      printf("You entered %d.\n", number);
   }

   printf("The if statement is easy.");

   return 0;
}
```

Output 1

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

Output 2

```
Enter an integer: 5
The if statement in C programming is easy.
```

If-else statement : The if-else statement is an extension of the simple if statement. The general form is. The if...else statement executes some code if the test expression is true (nonzero) and some other code if the test expression is false (0).

Syntax : if (condition)

```
{
    true statement;
}
else
{
    false statement;
}
statement-x;
```

If the condition is true , then the true statement and statement-x will be executed and if the condition is false, then the false statement and statement-x is executed.

Or

If test expression is true, codes inside the body of if statement is executed and, codes inside the body of else statement is skipped.

If test expression is false, codes inside the body of else statement is executed and, codes inside the body of if statement is skipped.

Flowchart

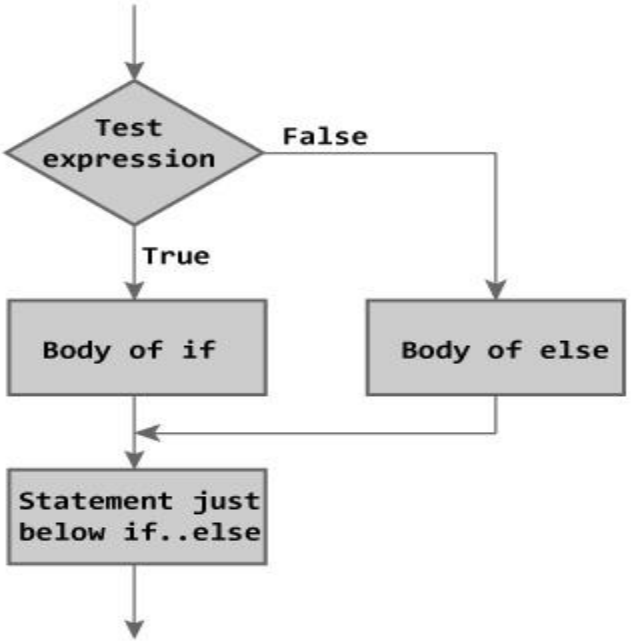


Figure: Flowchart of if...else Statement

Example:

// Program to check whether an integer entered by the user is odd or even

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

```
    int number;
    printf("Enter an integer: ");
    scanf("%d",&number);

    // True if remainder is 0
    if( number%2 == 0 )
        printf("%d is an even integer.",number);
    else
        printf("%d is an odd integer.",number);
    return 0;
}
```

Output

Enter an integer: 7
7 is an odd integer.

Nested if-else statement

When a series of decisions are involved, we may have to use more than on if-else statement in nested form. If –else statements can also be nested inside another if block or else block or both.

Syntax :

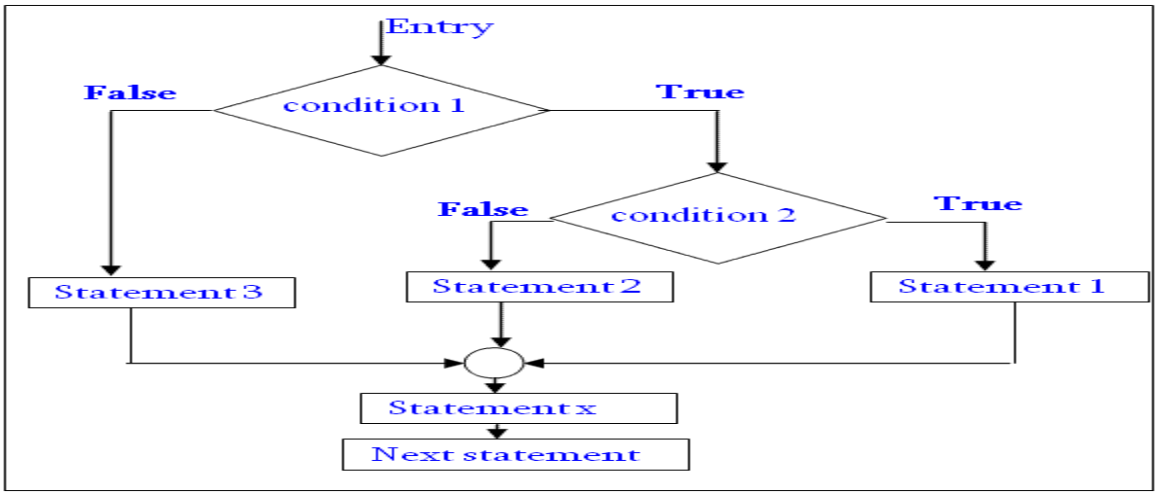
```
if(condition-1)
{                                   {
    if (condition-2)
    {
        statement-1;
    }
    else
    {
        statement-2;
    }
}
  else
  {
    statement-3;
```

}

statement-x;

If the condition-1 is false, the statement-3 and statement-x will be executed. Otherwise it continues to perform the second test. If the condition-2 is true, the true statement-1 will be executed otherwise the statement-2 will be executed and then the control is transferred to the statement-x

Flowchart



Example

```
#include<stdio.h>
int var1, var2;
printf("Input the value of var1:");
scanf("%d", &var1);
printf("Input the value of var2:");
scanf("%d",&var2);
if (var1 !=var2)
{
    printf("var1 is not equal to var2");
    //Below – if-else is nested inside another if block
    if (var1 >var2)
    {
       printf("var1 is greater than var2");
    }
    else
    {
       printf("var2 is greater than var1");
    }
}
else
```

```
{
    printf("var1 is equal to var2");
}
…
```

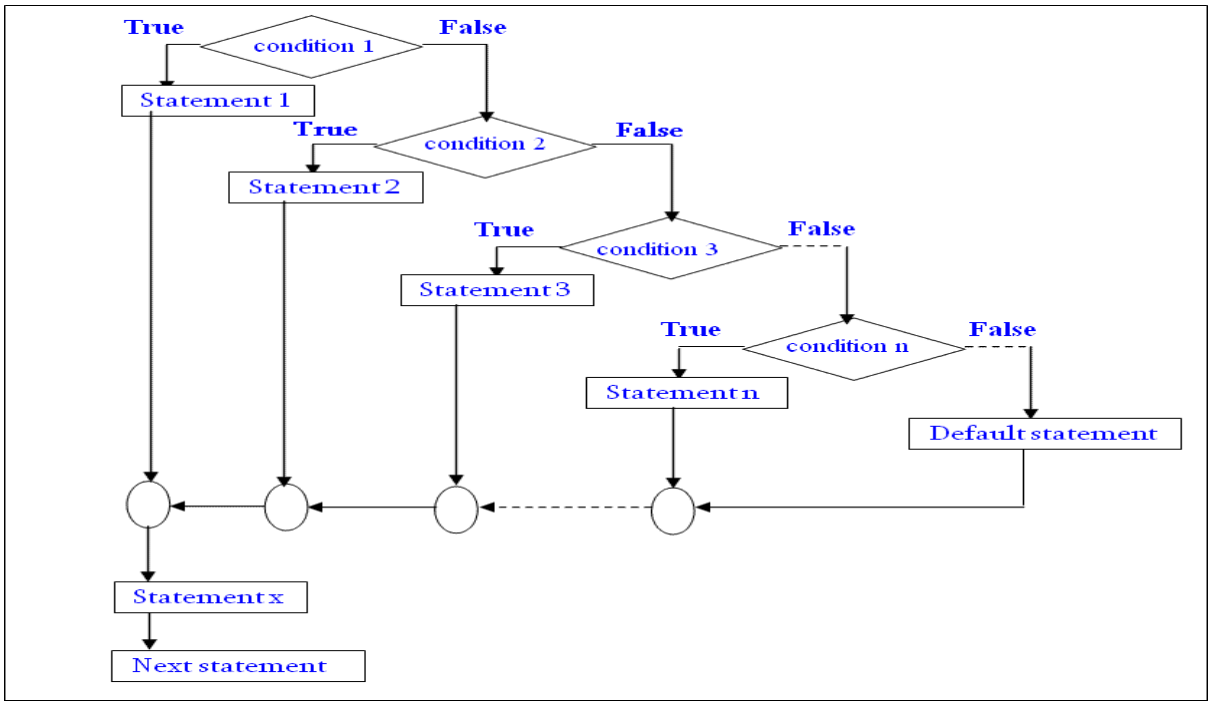
Else if ladder.

The if else-if statement is used to execute one code from multiple conditions**.**

Syntax : if(condition-1)

```
        {
            statement-1;
        }
    else if(condition-2)
      {
        statement-2;
       }
    else if(condition-3)
      {
           statement-3;
       }
    else if(condition-n)
     {
           statement-n;
       }
      else
       {
           default-statement;
         }
    statement-x;
```

Flowchart



Example

```
#include<stdio.h>
 #include<conio.h>
 void main(){
 int number=0;
 clrscr();
  printf("enter a number:");
 scanf("%d",&number);
 if(number==10){
 printf("number is equals to 10");
 }
 else if(number==50){
 printf("number is equal to 50");
 }
 else if(number==100){
 printf("number is equal to 100");
 }
 else{
 printf("number is not equal to 10, 50 or 100");
 }
 getch();
 }
```

Points to Remember

1. In *if* statement, a single statement can be included without enclosing it into curly braces `{ }`

```
int a = 5;

if(a > 4)

  printf("success");
```

   No curly braces are required in the above case, but if we have more than one statement inside *if* condition, then we must enclose them inside curly braces.

5. `==` must be used for comparison in the expression of *if* condition, if you use `=` the expression will always return true, because it performs assignment not comparison.
6. Other than **0(zero)**, all other values are considered as true.

```
if(27)

  printf("hello");
```

   In above example, hello will be printed.

Switch statement : when there are several options and we have to choose only one option from the available ones, we can use switch statement. Depending on the selected option, a particular task can be performed. A task represents one or more statements.

Syntax:

```
switch(expression)
{
 case value-1:
        statement/block-1;
        break;
 case value-2:
        statement/block t-2;
        break;
 case value-3:
        statement/block -3;
        break;
 case value-4:
        statement/block -4;
        break;
 default:
        default- statement/block t;
        break;
```

}

The expression following the keyword **switch** in any 'C' expression that must yield an integer value. It must be ab integer constants like 1,2,3 .

The keyword **case** is followed by an integer or a character constant, each constant in each must be different from all the other.

First the integer expression following the keyword **switch** is evaluated. The value it gives is searched against the constant values that follw the **case** statements. When a match is found, the program executes the statements following the case. If no match is found with any of the case statements, then the statements follwing the **default** are executed.

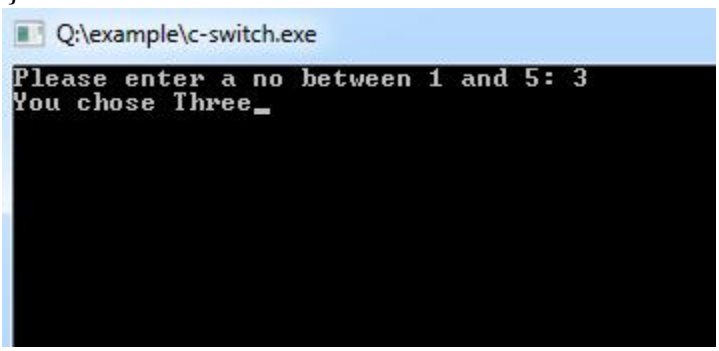
Rules for writing switch() statement.
1 : The expression in switch statement must be an integer value or a character constant.
2 : No real numbers are used in an expression.
3 : The default is optional and can be placed anywhere, but usually placed at end.
4 : The case keyword must terminate with colon (:).
5 : No two case constants are identical.
6 : The case labels must be constants.

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

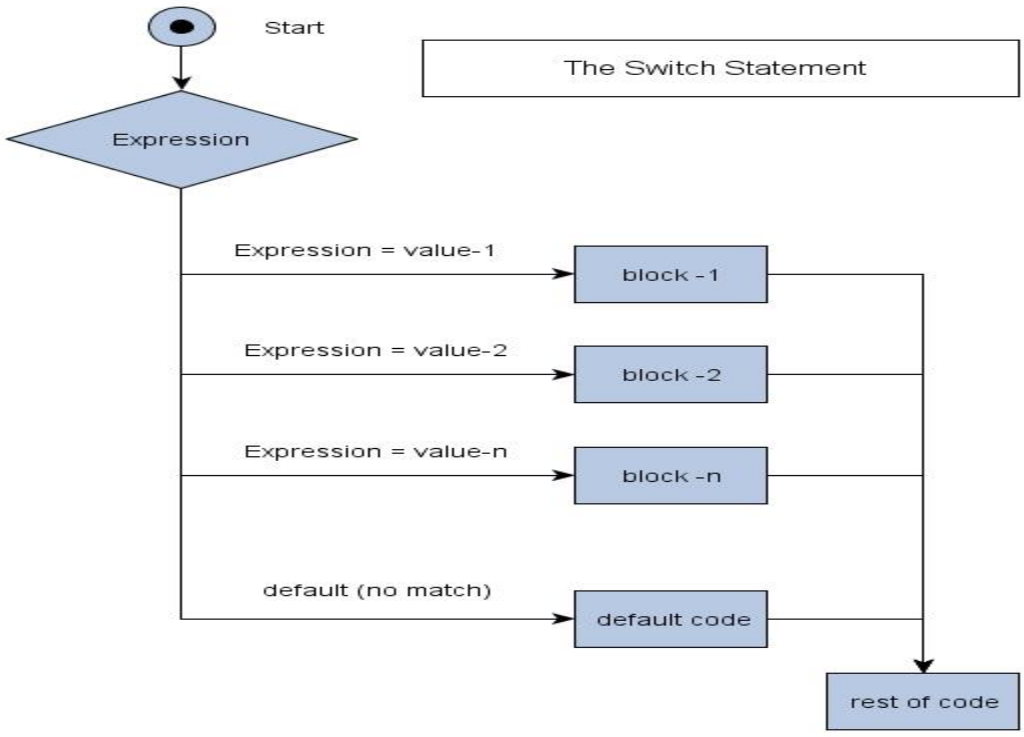
Example

```
#include<stdio.h>
 main()
{
int a;
printf("Please enter a no between 1 and 5: ");
scanf("%d",&a);
switch(a)
{
case 1:
printf("You chose One");
break;
case 2:
```

```
 printf("You chose Two");
break;
case 3:
 printf("You chose Three");
break;
case 4:
printf("You chose Four");
break;
 case 5: printf("You chose Five.");
 break;
default :
printf("Invalid Choice. Enter a no between 1 and 5"); break;
 }
 }
```



Flowchart



Points to Remember

It isn't necessary to use **break** after each block, but if you do not use it, all the consecutive block of codes will get executed after the matching block.

```
1.  int i = 1;
2.  switch(i)
3.  {
4.   case 1:
5.     printf("A");        // No break
6.   case 2:
7.     printf("B");        // No break
8.   case 3:
9.     printf("C");
10.    break;
11. }
```

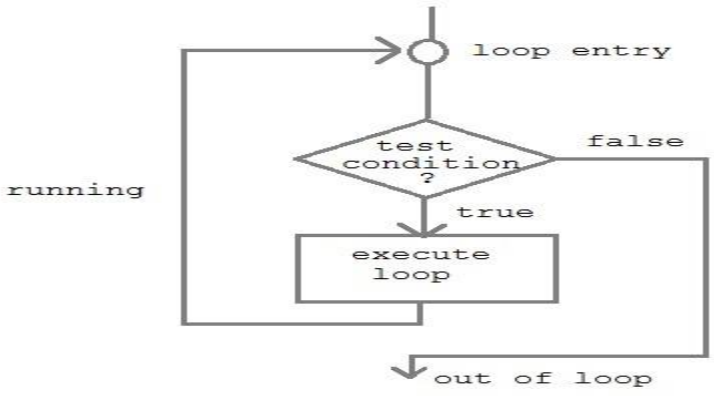
Output : A B C

The output was supposed to be only **A** because only the first case matches, but as there is no break statement after the block, the next blocks are executed, until the cursor encounters a break.

default case can be placed anywhere in the switch case. Even if we don't include the default case switch statement works.

Iteration Statements/ Loop Control Statements

How it Works



A sequence of statements are executed until a specified condition is true. This sequence of statements to be executed is kept inside the curly braces { } known as the Loop body. After every execution of loop body, condition is verified, and if it is found to be true the loop body is executed again. When the condition check returns false, the loop body is not executed.

The *loops in C language* are used to execute a block of code or a part of the program several times. In other words, it iterates/repeat a code or group of code many times**.**

Or Looping means a group of statements are executed repeatedly, **until some logical condition is satisfied**.

Why use loops in C language?

Suppose that you have to print table of 2, then you need to write 10 lines of code.By using the loop statement, you can do it by 2 or 3 lines of code only.

A looping process would include the following four steps.

1 : Initialization of a condition variable.

2 : Test the condition.

3 : Executing the body of the loop depending on the condition.

4 : Updating the condition variable.

C language provides three iterative/repetitive loops.

1 : while loop

2 : do-while loop

3 : for loop

While Loop: Syntax :

variable initialization ;

while (condition)

{

statements ;

variable increment or decrement ;

}

while loop can be addressed as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g int x=0;)
- condition(e.g while(x<=10))
- Variable increment or decrement (x++ or x-- or x=x+2)

The while loop is an entry controlled loop statement, i.e means the condition is evaluated first and it is true, then the body of the loop is executed. After executing the body of the loop, the condition is once again evaluated and if it is true, the body is executed once again, the process of repeated execution of the loop continues until the condition finally becomes false and the control is transferred out of the loop.

Example : Program to print first 10 natural numbers

```
#include<stdio.h>

#include<conio.h>

void main( )

{
```

```
int x;
x=1;
while(x<=10)
{
  printf("%d\t", x);
  x++;
}
getch();
}
```

Output

1 2 3 4 5 6 7 8 9 10

C Program to reverse number

```
#include<stdio.h>
#include<conio.h>
main()
{
int n, reverse=0, rem;
clrscr();
printf("Enter a number: ");
  scanf("%d", &n);
```

```
  while(n!=0)
  {
     rem=n%10;
     reverse=reverse*10+rem;
     n/=10;
  }
  printf("Reversed Number: %d",reverse);
getch();
}
```

Flowchart

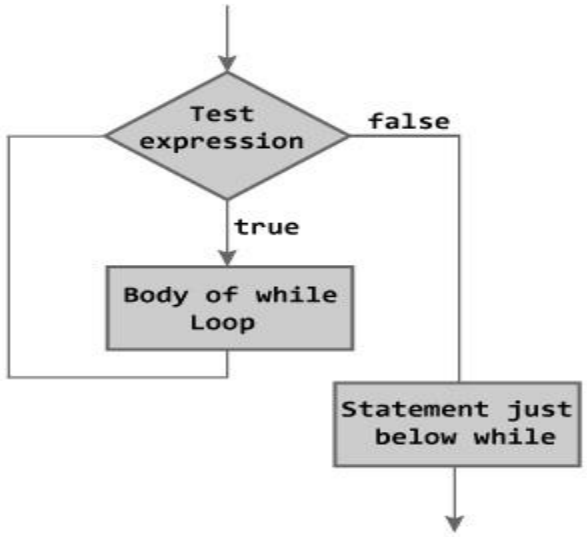


Figure: Flowchart of while Loop

do-while loop

Syntax : variable initialization ;

do{

statements ;

variable increment or decrement ;

}**while (condition);**

The **do-while** loop is an **exit controlled loop statement** The body of the loop are executed first and then the condition is evaluated. If it is true, then the body of the loop is executed once again. The process of execution of body of the loop is continued until the condition finally becomes false and the control is transferred to the statement immediately after the loop. The statements are always executed at least once.

Flowchart

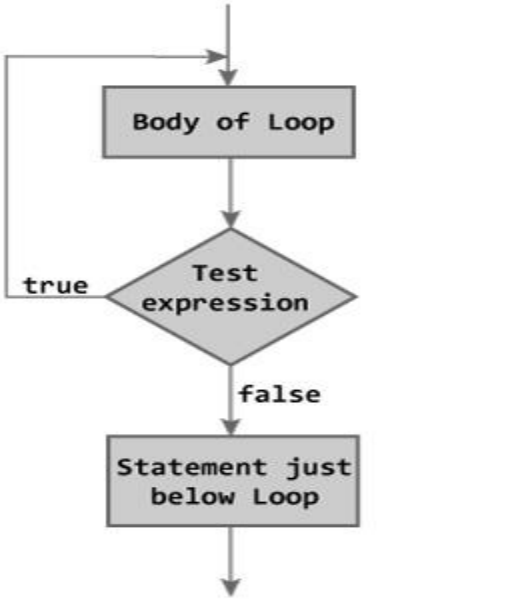


Figure: Flowchart of do...while Loop

Example : Program to print first ten multiple of 5

```
 #include<stdio.h>
#include<conio.h>
void main()
{
 int a,i;
 a=5;
 i=1;
 do
 {
   printf("%d\t",a*i);
   i++;
 }while(i <= 10);
getch();
}
```

Output

5 10 15 20 25 30 35 40 45 50

Example

```
main()
{
    int i=0
    do
    {
      printf("while vs do-while\n");
    }while(i= =1);
    printf("Out of loop");
}
```

Output:

while vs do-while

Out of loop

For Loop:

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

Syntax : for(initialization; condition; increment/decrement)

```
{
    Statements;
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
 int x;
 for(x=1; x<=10; x++)
 {
   printf("%d\t",x);
 }
 getch();
}
```

Output

1 2 3 4 5 6 7 8 9 10

Various forms of FOR LOOP

I am using variable num in all the below examples –

1) Here instead of num++, I'm using num=num+1 which is nothing but same as num++.

```
for (num=10; num<20; num=num+1)
```

2) Initialization part can be skipped from loop as shown below, the counter variable is declared before the loop itself.

```
int num=10;
for (;num<20;num++)
```

Must Note: Although we can skip init part but semicolon (;) before condition is must, without which you will get compilation error.

3) Like initialization, you can also skip the increment part as we did below. In this case semicolon (;) is must, after condition logic. The increment part is being done in for loop body itself.

```
for (num=10; num<20; )
{
    //Code
    num++;
}
```

4) Below case is also possible, increment in body and init during declaration of counter variable.

```
int num=10;
for (;num<20;)
{
    //Statements
    num++;
}
```

5) Counter can be decremented also, In the below example the variable gets decremented each time the loop runs until the condition num>10 becomes false.

```
for(num=20; num>10; num--)
```

Program to calculate the sum of first n natural numbers

```
#include <stdio.h>
int main()
{
   int num, count, sum = 0;
   printf("Enter a positive integer: ");
   scanf("%d", &num);
   // for loop terminates when n is less than count
```

```
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }
    printf("Sum = %d", sum);
    return 0;
}
```

Output

```
Enter a positive integer: 10
Sum = 55
```

Factorial Program using loop

```
#include<stdio.h>
#include<conio.h>
void main(){
  int i,fact=1,number;
  clrscr();
  printf("Enter a number: ");
  scanf("%d",&number);

  for(i=1;i<=number;i++){
     fact=fact*i;
  }
  printf("Factorial of %d is: %d",number,fact);
  getch();
}
```

Output:

Enter a number: 5

Factorial of 5 is: 120

Flow Chart of *for* Loop :

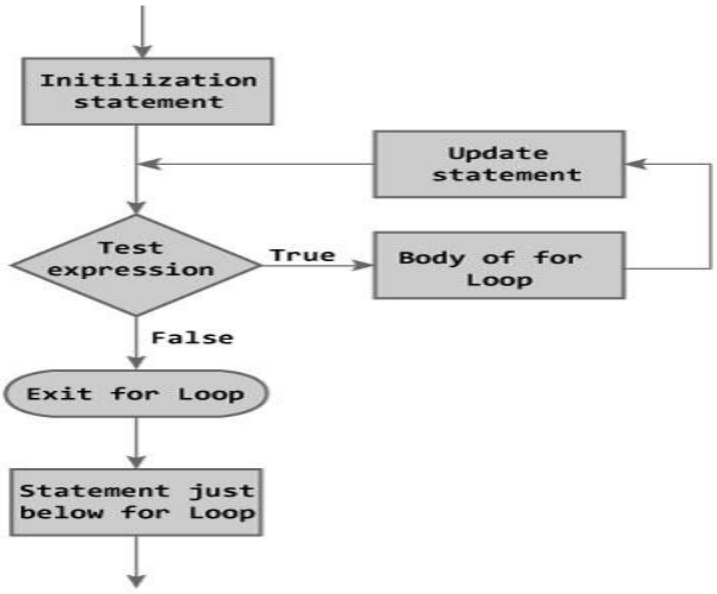
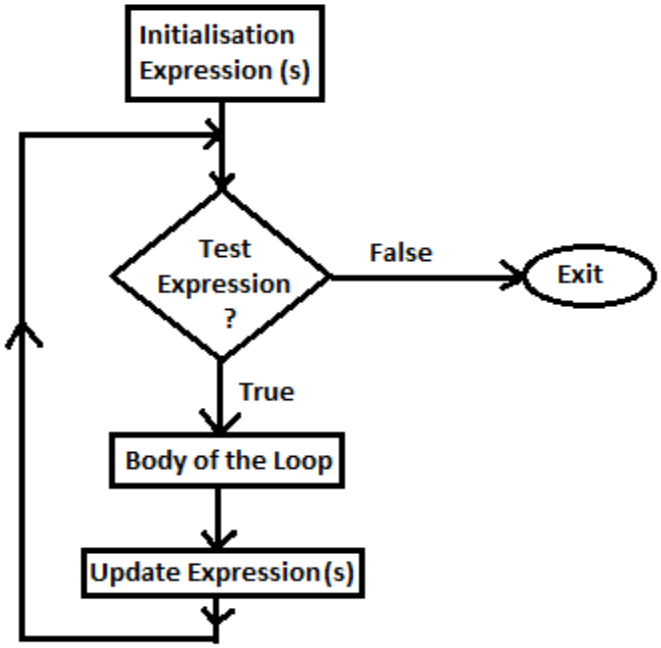


Figure: Flowchart of for Loop



Infinitive for loop in C

If you don't initialize any variable, check condition and increment or decrement variable in for loop, it is known as infinitive for loop. In other words, if you place 2 semicolons in for loop, it is known as infinitive for loop.

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

```
printf("infinitive for loop example by javatpoint");
}
```

Basis of Difference	***For* Loop**	***While* Loop**	***Do While* Loop**
Where to Use *for* Loop, *while* Loop and *do while* Loop	The *for* loop is appropriate when we know in advance how many times the loop will be executed.	The other two loops i.e. *while and do while loops* are more suitable in the situations where it is not known before hand when the loop will terminate.	
		In case if the test condition fails at the beginning, and you may not want to execute the body of the loop even once if it fails, then the while loop should be preferred.	In case if the test condition fails at the beginning, and you may want to execute the body of the loop atleast once even in the failed condition, then the do while loop should be preferred.
How all the three loops works?	A for loop initially initiates a counter variable (initialization-expression), then it checks the test-expression, and executes the body of the loop if the test expression is true. After executing the body of the loop,	A while loop will always evaluate the test-expression initially. It the test-expression becomes true, then the body of the loop will be executed. The update	A do while loop will always executed the code in the do {} i.e. body of the loop block first and then evaluates the condition. In this case also, the counter variable is initialized outside the body of the loop.

	the update-expression is executed which updates the value of counter variable.	expression should be updated inside the body of the while. However, the counter variable is initialized outside the body of the loop.	
Position of the statements : • Initialization • test-expression • update-expression	In for loop, all the three statements are placed in one position	In while and do while loop, they are placed in different position.	
Syntax of Loops	**for (** **initialization-exp.(s);** **test-expression(s);** **update-expression(s)** **)** **{** **body-of-the-loop;** **}**	**while(test-expression)** **{** **body-of-the-loop;** **update-expression(s);** **}**	**do {** **body-of-the-loop;** **update-expression(s);** **}** **while (test-expression);**

Which one is Entry Controlled Loop and Which one is Exit Controlled Loop ?	Both loops i.e. for loop and while loop are entry controlled loop, means condition is checked first and if the condition is true then the body of the loop will executes.		do while loop is an exit controlled loop, means means that condition is placed after the body of the loop and is evaluated before exiting from the loop.
Conversion of one Loop to another Loop or Example : Print numbers from 1 to 10 using all the three loops.	**:** **:** **for (int i=1; i<=10; i++)** **{** **Printf("%d",i); }**	**int i = 1;** **:** **:** **while (i<=10)** **{** **Printf("%d",i); ++i** **}**	**int i = 1;** **:** **:** **do** **{** **Printf("%d",i); ++i;** **}** **while (i<=10)**

Nested for loop

We can also have nested **for** loops, i.e one **for** loop inside another **for** loop. nesting is often used for handling multidimensional arrays.

Syntax:

for(**initialization**; **condition**; **increment/decrement**)

{

for(**initialization**; **condition**; **increment/decrement**)

```
    {
      statement ;
    }
}
```

Example:

```
main()
{
    for (int i=0; i<=5; i++)
    {
        for (int j=0; j<=5; j++)
        {
            printf("%d, %d",i ,j);
        }
   }
}
```

Example : Program to print half Pyramid of numbers

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

```
  for(j=i;j>0;j--)
  {
    printf("%d",j);
  }
 }
 getch();
}
```

Output

1

21

321

4321

54321

Jump Statements

Jumping statements are used to transfer the program's control from one location to another, these are set of keywords which are responsible to transfer program's control within the same block or from one function to another**.**

There are four jumping statements in C language:

- goto statement
- return statement
- break statement
- continue statement

goto statement : goto statement doesnot require any condition. This statement passes control anywhere in the program i.e, control is transferred to another part of the program without testing any condition.

Syntax : goto label;

. . . . .

. . . . .

label:

statements;

Inthissyntax, **label** isan 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.

Or

The goto statement requires a label to identify the place to move the execution. A label is a valid variable/identifier name and must be ended with colon **(:)**

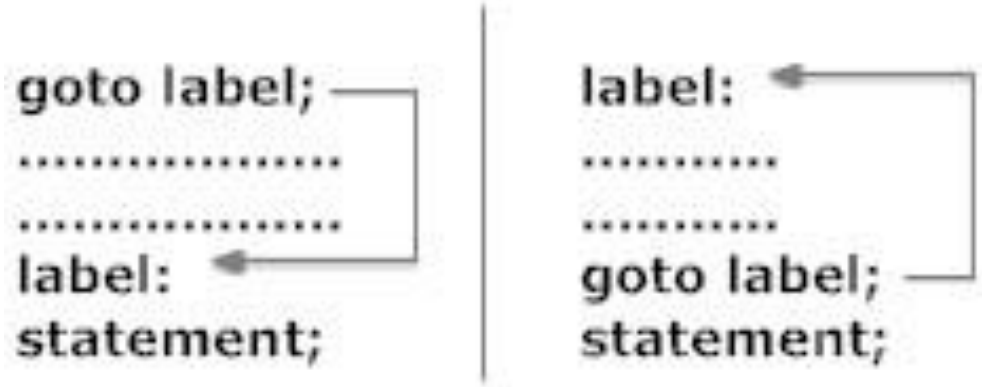
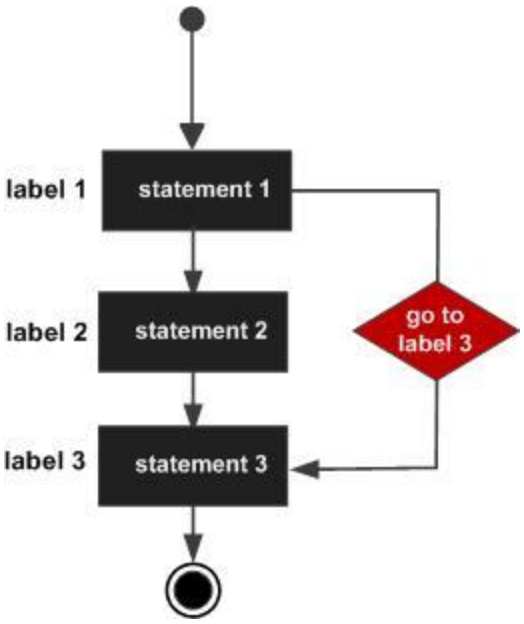


Figure: Working of goto statement

Flowchart



Example

```
int 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;
  return 0;
```

}

Output

Enter you age:19

you are eligible for voting

Enter you age:15

you are not eligible to vote

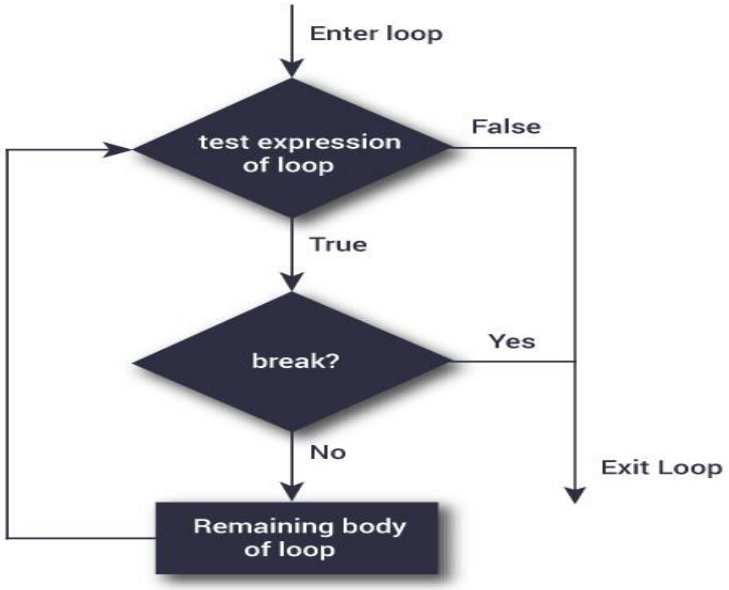
Break Statement

Break is a keyword. The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used/ associated with decision making statement such as if ,if-else.

Syntax of break statement

```
break;
```

Flowchart



How break statement works?

```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

Example

```
#include <stdio.h>

#include <conio.h>

 void main(){

int i=1;//initializing a local variable

clrscr();

  //starting a loop from 1 to 10

for(i=1;i<=10;i++){
```

```
printf("%d \n",i);
if(i==5){//if value of i is equal to 5, it will break the loop
break;
}
}//end of for loop
getch();
}
```

Output

12345

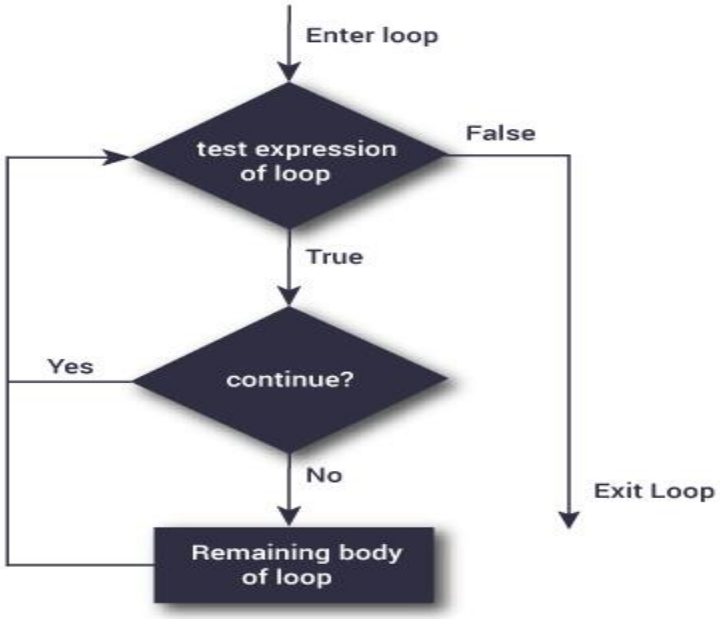
Continue Statement

Continue is keyword exactly opposite to break. The continue statement is used for continuing next iteration of loop statements. When it occurs in the loop it does not terminate, but it skips some statements inside the loop / the statements after this statement. . The continue statement is used/ associated with decision making statement such as if ,if-else.

Syntax of continue Statement

```
continue;
```

Flowchart of continue Statement



How continue statement works?

```
while (test Expression)
{
      // codes
      if (condition for continue)
      {
            continue;
      }
      // codes
}
```

```
for (init, condition, update)
{
      // codes
      if (condition for continue)
      {
            continue;
      }
      // codes
}
```

Example

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main(){
4. int i=1;//initializing a local variable
5. clrscr();
6. //starting a loop from 1 to 10
7. for(i=1;i<=10;i++){
8. if(i==5){//if value of i is equal to 5, it will continue the loop
9. continue;
10. }
11. printf("%d \n",i);
12. }//end of for loop
13.  getch();
14. }
```

Output

1234678910

Comparision between break and continue statements

Break	Continue
1 : break statement takes the control to the ouside of the loop	1 :continue statement takes the control to the beginning of the loop..
2 : it is also used in switch statement.	2 : This can be used only in loop statements.
3 : Always associated with if condition in loops.	3 : This is also associated with if condition.

ARRAYS

Using Arrays in C

C supports a derived data type known as *array* that can be used to handle large amounts of data (multiple values) at a time**.**

Definition:

An array is a group (or collection) of same data types.

Or

An array is a collection of data that holds fixed number of values of same type.

Or

Array is a *collection* or *group* of elements (data). All the elements of array are *homogeneous* (similar). **It has contiguous memory location**.

Or

An array is a data structured that can store a fixed size sequential collection of elements of same data type.

What's the need of an array?

Suppose you have to store marks of 50 students, one way to do this is allotting 50 variables. So it will be typical and hard to manage. For example we can not access the value of these variables with only 1 or 2 lines of code.

Another way to do this is array. By using array, we can access the elements easily. Only few lines of code is required to access the elements of array.

Where arrays are used

- to store list of Employee or Student names,
- to store marks of a students,
- or to store list of numbers or characters etc.

Advantage of C Array

1) Code Optimization: Less code to the access the data.

2) Easy to traverse data: By using the for loop, we can retrieve the elements of an array easily.

3) Easy to sort data: To sort the elements of array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of Array

Fixed Size: Whatever size, we define at the time of declaration of array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList

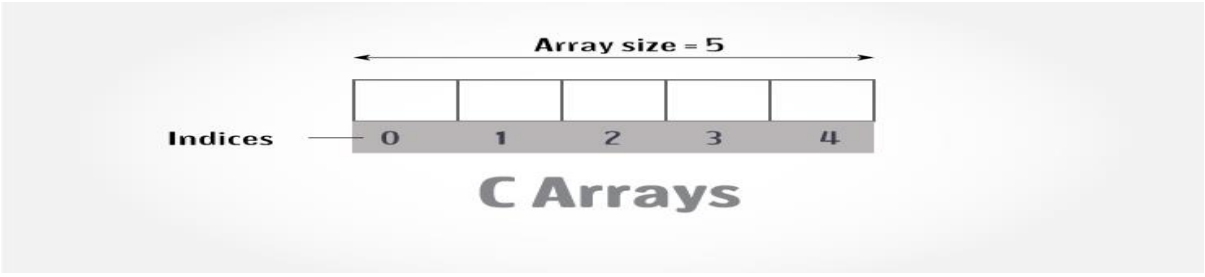
Declaration of an Array

data-type variable-name[size/length of array];

For example:

int arr[10];

int arr[5];



Here **int** is the data type, **arr** is the name of the array and 10 is the size of array. It means array **arr** can only contain 10 elements of **int** type. **Index** of an array starts from 0 to size-1 i.e first element of **arr** array will be stored at arr[0] address and last element will occupy arr[9].

Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value). An array can be initialized at either **compile time** or at **runtime**.

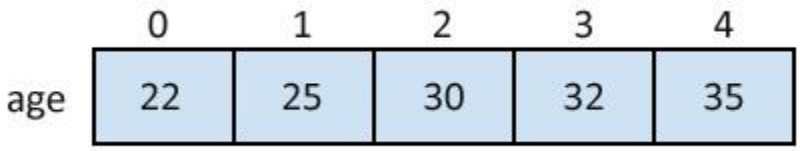
Compile time Array initialization

Compile time initializtion of array elements is same as ordinary variable initialization.

Syntax : data_type array_name[size]={v1,v2,…vn/list of values ;

Example

int age[5]={22,25,30,32,35};



int marks[4]={ 67, 87, 56, 77 }; *//integer array initialization*

float area[5]={ 23.4, 6.8, 5.5 }; *//float array initialization*

int marks[4]={ 67, 87, 56, 77, 59 }; //Compile time error

Different ways of initializing arrays :

1 : Initilizing all specified memory locations

2 : Partial array initialization.

3 : Intilization without size.

4 : String initialization.

1 : Initilizing all specified memory locations : If the number of values to be initialized is equal to size of array. Arrays can be initialized at the time of declaration. Array elements can be initialized with data items of type int,float,char, etc.

Ex : consider integer initialization

int a[5]={10,20,30,40,50};

During compilation, 5 contiguous memory locations are reserved by the compiler for the variable a and all these locations are initialized.

The array a is initialized as

a[0]	a[1]	a[2]	a[3]	a[4]
10	20	30	40	50
1000	1002	1004	1006	1008

If the size of integer is 2 bytes, 10 bytes will be allocated for the variable a.

Ex : consider character initialization

char b[8] = {'C','O','M','P','U','T','E','R'};

The array b is initialized as

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]
C	O	M	P	U	T	E	R

Other Examples : char b[5]={'J','B','R','E','C','B'};

//error : number of initial values are more than the size of array.

Other Example : int a[5]={10,20,30,40,50,60};

//error : Number of initial values are more than the size of array.

2 : Partial Array Initilization : partial array initialization is possible in C language. If the number of values to be initialized is less than the size of the array, then the elements are initialized in the order from 0^{th} location. The remaining locations will be initialized to zero automatically.

Ex : Consider the partial initilization

```
int a[5]={10,15};
```

Eventhough compiler allocates 5 memory locations, using this declaration statement, the compiler initializes first two locations with 10 and 15, the next set of memory locations are automatically initialized to zero.

The array a is partial initialization as

a[0]	a[1]	a[2]	a[3]	a[4]
10	15	0	0	0
1000	1002	1004	1006	1008

How to access the elements of an array?

You can access elements of an array by **indices/index.** You can use array subscript (or index) to access any element stored in array. Subscript starts with 0, which means array_name[0] would be used to access first element in an array.

In general array_name[n-1] can be used to access nth element of an array. where n is any integer number.

Example

float mark[5];

Suppose you declared an array mark as above. The first element is mark[0], second element is mark[1] and so on.

Few key notes:

- Arrays have 0 as the first index not 1. In this example, mark[0]
- If the size of an array is n, to access the last element, (n-1) index is used. In this example, mark[4]
- Suppose the starting address of mark[0] is 2120d. Then, the next address, a[1], will be 2124d, address of a[2] will be 2128d and so on. It's because the size of a float is 4 bytes.

Input data into array

As you can see, in above example that I have used 'for loop' and 'scanf statement' to enter data into array. You can use any loop for data input.

Code:

```
for (x=0; x<=19;x++)

{

    printf("enter the integer number %d\n", x);

    scanf("%d", &num[x]);

}
```

Reading out data from an array

For example you want to read and display array elements, you can do it just by using any loop. Suppose array is mydata[20].

```
for (int i=0; i<20; i++)

{

   printf("%d\n", mydata[x]);

}
```

Exmaple

```
#include<stdio.h>

#include<conio.h>

void main()

{

 int i;

 int arr[]={2,3,4};    //Compile time array initialization

 for(i=0 ; i<3 ; i++) {

    printf("%d\t",arr[i]);

 }

 getch();

}
```

Output

2 3 4

Exmaple

```
1.  include <stdio.h>
2.  #include <conio.h>
3.  void main(){
4.  int i=0;
5.  int marks[5]={20,30,40,50,60};//declaration and initialization of array
6.  clrscr();
7.
8.  //traversal of array
9.  for(i=0;i<5;i++){
10. printf("%d \n",marks[i]);
11. }
12.
13. getch();
14. }
```

Output

```
20
30
40
50
60
```

Runtime Array initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values.

Example

```
#include<stdio.h>

#include<conio.h>

void main()

{

 int arr[4];

 int i, j;

 printf("Enter array element");

 for(i=0;i<4;i++)

 {

  scanf("%d",&arr[i]);    //Run time array initialization

 }

 for(j=0;j<4;j++)

 {

  printf("%d\n",arr[j]);

 }

 getch();
```

}

Two-Dimensional Arrays

The two dimensional array in C language is represented in the form of rows and columns, also known as matrix. It is also known as *array of arrays* or *list of arrays*.

The two dimensional, three dimensional or other dimensional arrays are also known as *multidimensional* arrays.

Declaration of two dimensional Array

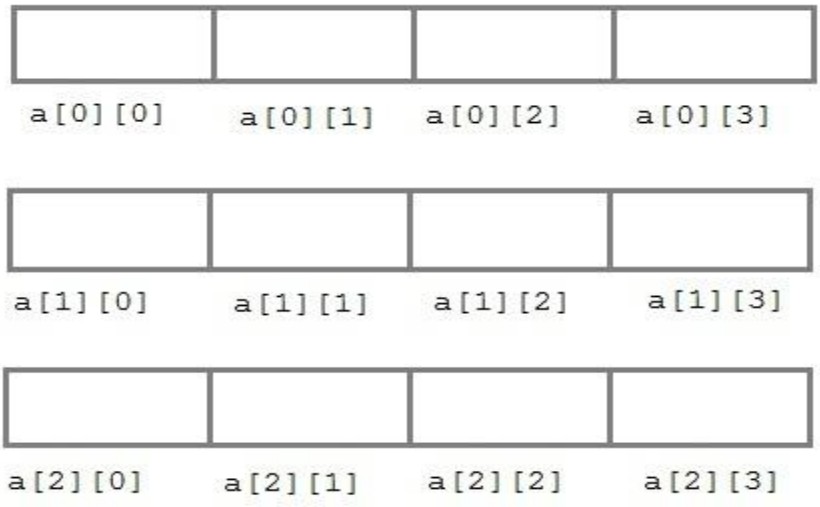
data_type array_name[size1][size2];

Example

int twodimen[4][3];

Example :

int a[3][4];



Initialization of 2D Array

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

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.

Example

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main(){
4. int i=0,j=0;
5. int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
6. clrscr();
7. //traversing 2D array
8. for(i=0;i<4;i++){
9.  for(j=0;j<3;j++){
10.   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
11.  }//end of j
12. }//end of i
13.  getch();
14. }
```

Output

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

Example Write a C program Addition of Two Matrices

```
#include<stdio.h>

#include<conio.h>
```

```
void main()
{
 int a[25][25],b[25][25],c[25][25],i,j,m,n;
 clrscr();
 printf("enter the rows and colums of two matrics:\n");
 scanf("%d%d",&m,&n);
 printf("\nenter the elements of A matrics");
 for(i=0;i<m;i++)
 {
  for(j=0;j<n;j++)
  scanf("\t%d",&a[i][j]);
 }
 printf("\nenter the elements of B matrics");
 for(i=0;i<m;i++)
 {
  for(j=0;j<n;j++)
  scanf("\t%d",&b[i][j]);
 }
 printf("\nThe elements of A matrics");
 for(i=0;i<m;i++)
 {
  printf("\n");
  for(j=0;j<n;j++)
```

```
  printf("\t%d",a[i][j]);
 }
 printf("\nThe elements of B matrics");
 for(i=0;i<m;i++)
 {
  printf("\n");
  for(j=0;j<n;j++)
  printf("\t%d",a[i][j]);
 }
 printf("\nThe additon of two matrics");
 for(i=0;i<m;i++)
 {
  printf("\n");
  for(j=0;j<n;j++)
  {
    c[i][j]=a[i][j]+b[i][j];
    printf("\t%d",c[i][j]);
  }
 }
 getch();
}
```

Write a C program Multiplication of Two Matrices.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a[25][25],b[25][25],c[25][25],i,j,m,n,k,r,s;
  clrscr();
  printf("enter the rows and colums of A matrics:\n");
  scanf("%d%d",&m,&n);
  printf("enter the rows and colums of B matrics:\n");
  scanf("%d%d",&r,&s);
  printf("\nenter the elements of A matrics");
  for(i=0;i<m;i++)
  {
    for(j=0;j<n;j++)
    scanf("\t%d",&a[i][j]);
  }
  printf("\nenter the elements of B matrics");
  for(i=0;i<m;i++)
  {
    for(j=0;j<n;j++)
    scanf("\t%d",&b[i][j]);
```

```
}
printf("\nThe elements of A matrics");
for(i=0;i<m;i++)
{
  printf("\n");
  for(j=0;j<n;j++)
  printf("\t%d",a[i][j]);
}
printf("\nThe elements of B matrics");
for(i=0;i<m;i++)
{
  printf("\n");
  for(j=0;j<n;j++)
  printf("\t%d",b[i][j]);
}

for(i=0;i<m;i++)
{
  printf("\n");
  for(j=0;j<n;j++)
  {
     c[i][j]=0;
     for(k=0;k<m;k++)
```

```
        c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
  }

  printf("\nThe Multiplication of two matrics");
  for(i=0;i<m;i++)
  {
    printf("\n");
    for(j=0;j<n;j++)
    printf("\t%d",c[i][j]);
  }
   getch();
}
```

Multidimensional Arrays

How to initialize a multidimensional array?

Initialization of a three dimensional array.

You can initialize a three dimensional array in a similar way like a two dimensional array. Here's an example

```
int test[2][3][4] = {
                { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
              };
```

Example

```
#include <stdio.h>
int main()
{
   // this array can store 12 elements
   int i, j, k, test[2][3][2];
   printf("Enter 12 values: \n");
   for(i = 0; i < 2; ++i) {
      for (j = 0; j < 3; ++j) {
         for(k = 0; k < 2; ++k ) {
            scanf("%d", &test[i][j][k]);
         }
      }
   }
   // Displaying values with proper index.
   printf("\nDisplaying values:\n");
   for(i = 0; i < 2; ++i) {
      for (j = 0; j < 3; ++j) {
         for(k = 0; k < 2; ++k ) {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
         }
      }
   }
   return 0;
}
```

Output

Enter 12 values:

123456789101112

Displaying Values:

test[0][0][0] = 1

test[0][0][1] = 2

test[0][1][0] = 3

test[0][1][1] = 4

test[0][2][0] = 5

test[0][2][1] = 6

test[1][0][0] = 7

test[1][0][1] = 8

test[1][1][0] = 9

test[1][1][1] = 10

test[1][2][0] = 11

test[1][2][1] = 12

STRINGS:

String Concepts

String is an *array of characters* that is terminated by \0 (null character). This null character indicates the end of the string. Strings are always enclosed by double quotes (" "). Whereas, character is enclosed by single quotes.

Or

In ‘C’ language the group of characters, digits, and symbols enclosed within double quotation (" ") marks are called as string otherwise a string is an array of characters and terminated by NULL character which is denoted by the escape sequence ‘\0’.

C Strings

Declaration of String: C does not support string as a data type. However, it allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and it is always declared as an array of characters.

The general form of declaration of a string variable is :

Syntax: char string_name[size];

The size determines the number of characters in the string name.

Note: In declaration of string size must be required to mention otherwise it gives an error.

Ex: char str[]; // Invalid

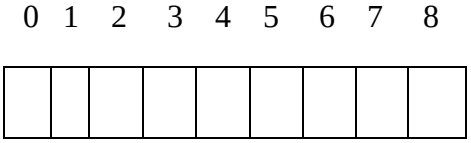
char str[0]; // Invalid

char str[-1]; // Invalid

char str[10]; // Valid

char a[9]; //Valid

Using this declaration the compiler allocates 9 memory locations for the variable a ranging from 0 to 8.



Here, the string variable a can hold maximum of 9 characters including NULL(\0) character.

Initializing Array string

Syntax : char string_name[size]={“string” };

Note: In Initialization of the string if the specific number of character is not initialized it then rest of all character will be initialized with NULL.

char str[5]={'5','+','A'};

str[0]; ---> 5

str[1]; ---> +

str[2]; ---> A

str[3]; ---> NULL

str[4]; ---> NULL

Note: In initialization of the string we can not initialized more than size of string elements.

Ex:

char str[2]={'5','+','A','B'}; // Invalid

Different ways of initialization can be done in various ways :

1 : Initilizing locations character by character.

2 : Partial array initialization.

3 : Intilization without size.

4 : Array initialization with a string .

1 : Initilizing locations character by character

Consider the following declaration with initialization,

Char b[9]={'C','O','M','P','U','T','E','R'};

The compiler allocates 9 memory locations ranging from 0 to 8 and these locations are initialized with the characters in the order specified. The remaining locations are automatically initialized to null characters.

C	O	M	P	U	T	E	R	\0
0	1	2	3	4	5	6	7	8

2 : Partial Array Initilization : If the characters to be initialized is less than the size of the array, then the characters are stored sequentially from left to right.The remaining locations will be initialized to NULL characters automatically.

Ex : Consider the partial initilization

int a[10]={'R','A','M','A' };

The compiler allocates 10 bytes for the variable a ranging from 0 to 9 and initializes first four locations with the ASCII characters of 'R', 'A', 'M', 'A'.The remaining locations are automatically filled with NULL characters (i.e,\0).

R	A	M	A	\0	\0	\0	\0	\0	\0
0	1	2	3	4	5	6	7	8	9

3 : Initilization without size : consider the declaration along with the initialization

char b[]={'C','O','M','P','U','T','E','R'};

In this declaration, The compiler will set the array size to the total number of initial values i.e 8. The character will be stored in these memory locations in the order specified.

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]
C	O	M	P	U	T	E	R

4) Array Initilization with a String : consider the declaration with string initialization.

char b[ ] = "COMPUTER";

Here, the string length is 8 bytes. But , string size is 9 bytes. So the compiler reserves 8+1 memory locations and these locations are initialized with the characters in the order specified. The string is terminated by \0 by the compiler.

C	O	M	P	U	T	E	R	\0
0	1	2	3	4	5	6	7	8

The string "COMPUTER" contin 8 charactes, because it is a string. It always ends with null character. So, the array is 9 bytes (i.e string length+1 byte for null character).

Reading and Writing Strings : The '%s' control string can be used in scanf() statement to read a string from the teriminal and the same may be used to write string to the terminal in printf() statement.

Example : char name[10];

scanf("%s",name);

printf("%s",name);

Example:

```
1. #include <stdio.h>
2. void main ()
3. {
4.    char ch[13]={'c', 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', i', 'n', 'g', '\0'};
5.    char ch2[13]="cprogramming";
6.
7.    printf("Char Array Value is: %s\n", ch);
8.    printf("String Literal Value is: %s\n", ch2);
9. }
```

Output

Char Array Value is: cprogramming

String Literal Value is: cprogramming

Example:

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
```

```
    printf("Your name is %s.", name);
    return 0;
}
```

Output

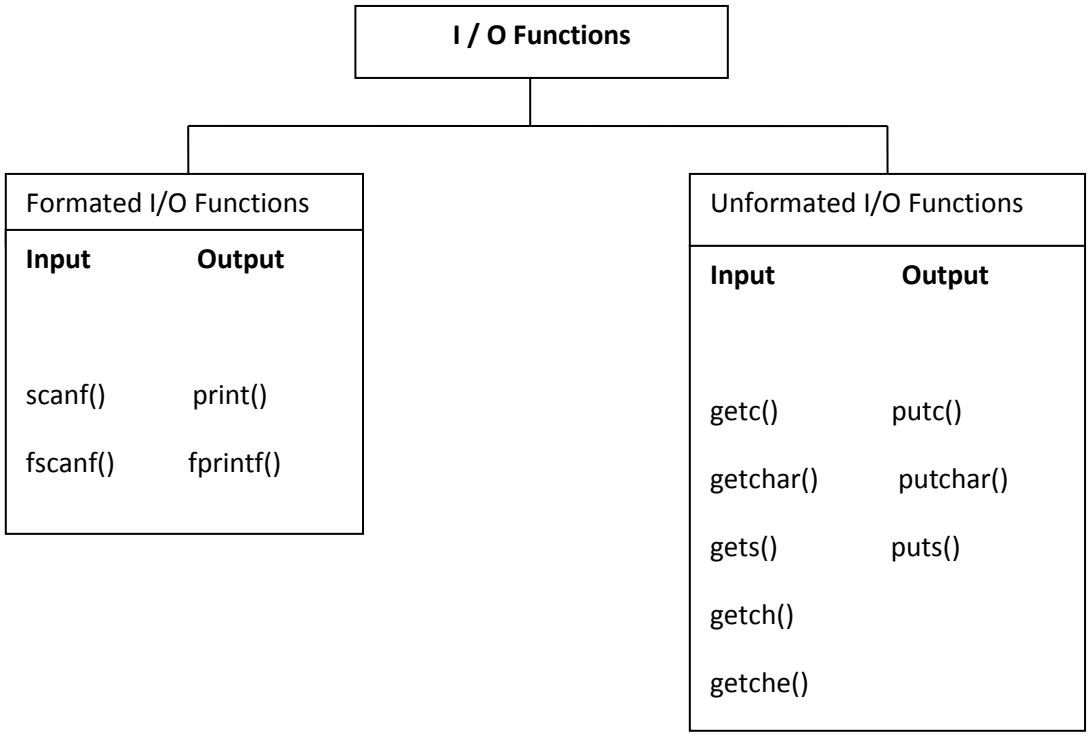
Enter name: Dennis Ritchie

Your name is Dennis.

String Input/output Functions

The strings can be read from the keyboard and can be displayed onto the monitor using various functions.

The various input and output functions that are associated with can be classified as



Unformated I/O Functions

1 : getchar() function : A single character can be given to the computer using 'C' input library function getchar().

Syntax : char variable=getchar();

The getchar() function is written in standared I/O library. It reads a single character from a standared input device. This function do not require any arguments, through a pair of parantheses, must follow the statements getchar().

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  char ch;
  clrscr();
  printf("Enter any character/digit:");
  ch=getchar();
  if(isalpha(ch)>0)
   printf("it is a alphabet:%c\n",ch);
  else if(isdigit(ch)>0)
   printf("it is a digit:%c\n",ch);
  else
   printf("it is a alphanumeric:%c\n",ch);
   getch();
```

}.

OUTPUT : Enter any character/Digit : abc

it is a alphabet:a

2 : putchar() function : The putchar() function is used to display one character at a time on the standared output device. This function does the reverse operation of the single character input function.

Syntax : putchar(character varaiable);

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  char ch;
  printf("Enter any alphabet either in lower or uppercase:");
  ch=getchar();
  if(islower(ch))
   putchar(toupper(ch));
  else
   putchar(tolower(ch));
  getch();
}
```

OUTPUT : Enter any alphabet either in lower or uppercase :a

A

3 : gets() : The gets() function is used to read the string (String is a group of characters) from the standard input device (keyboard).

Syntax : gets(char type of array variable);

Ex :

```
#include<stdio.h>
 #include<conio.h>
 void main()
{
  char str[40];
  clrscr();
   printf("Enter String name:");
    gets(str);
    printf("Print the string name%s:",str);
    getch();
}
```

OUTPUT : Enter the string : reddy

Print the string :reddy

4 : puts() : The puts() function is used to display the string to the standared output device (Monitor).

Syntax : puts(char type of array variable);

Program using gets() function and puts() function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 char str[40];
```

```
puts("Enter String name:");
 gets(str);
 puts("Print the string name:");
 puts(str);
 getch();
}
```

OUTPUT : Enter string name :

subbareddy

Print the string name

subbareddy

getch() function :The getch function reads a single character directly from the keyboard, without echoing to the screen.

Syntax : int getch();

Ex :

```
#include<stdio.h>
    void main()
    {
        char c;
        c=getch();
    }
```

getche() function : The getche() function reads a single character from the keyboard and echoes it to the current text window.

Syntax : int getche();

Ex :

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

```
    char c;
    c=getche();
}
```

getc() function : This function is used to accept a single character from the standared input to a character variable.

Syntax : character variable=getc();

putc() function : This function is used to display a single character in a character variable to standared output device.

Syntax : putc(character variable);

Array of Strings

String Manipulation Functions/ String Handling Functions

The various string handling functions that are supported in C language are as shown

String Function	Description
strlen(str)	Returns the length of the string str.
strcpy(str1,str2)	Copies the string str2 to string str1
strcat(str1,str2)	Append string str2 to string str1.
strlwr(str)	Converts the string str to lowercase
strupr(str)	Converts the string str to uppercase.
strrev(str)	Reverse the string str.
strcmp(str1,str2)	Compare two strings str1 and str2.

All these functions are defined in **string.h header file.**

1 : strlen(string) – String Length : This function is used to count and return the number of characters present in a string.

Syntax : var=strlen(string);

Ex : Progrm using strlen() function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  char name[]="JBREC";
  int len1,len2;
  clrscr();
  len1=strlen(name);
  len2=strlen("JBRECECE");
  printf("The string length of %s is: %d\n",name,len1);
  printf("The string length of %s is: %d","JBRECECE",len2);
  getch();
}
```

OUTPUT :

The string length of JBREC is : 5

The string length of JBRECECE is :8

Write a program to find the length of string

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

```
  char str[10];
  int index;
  printf("Enter the string:");
  scanf("%s",str);
  for(index=0;str[index]!=0;index++);
  printf("The length of string is:%d",index);
  getch();
}
```

OUTPUT :

Enter the string : subbareddy

The length of string is :10

2 : strcpy(string1,string2) – String Copy : This function is used to copy the contents of one string to another string.

Syntax : strcpy(string1,string2);

Where

string1 : is the destination string.

string 2: is the source string.

i.e the contents of string2 is assigned to the contents of string1.

Ex : Progrm using strcpy() function

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

```
 char str1[]="REDDY";
 char str2[10];
 strcpy(str2,str1);
 printf("The string1 is :%s\n",str1);
 printf("The string2 is :%s\n",str2);
 strcpy(str2,str1+1);
 printf("The string1 is :%s\n",str1);
 printf("The string2 is :%s",str2);
}
```

OUTPUT :

The string1 is : REDDY

The string2 is : REDDY

The string1 is : REDDY

The string2 is : EDDY

//Write a program to copy contents of one string to another string.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char str1[10],str2[20];
  int index;
  printf("Enter the string\n");
  scanf(“%s”,str1);
  for(index=0;str1[index]!='\0';index++)
```

```
        str2[index]=str1[index];
        str2[index]='\0';
        printf("String1 is :%s\n",str1);
        printf("String2 is :%s\n",str2);
        getch();
}
```

OUTPUT :

Enter the string : cprogramming

String1 is : cprogramming

String2 is : cprogramming

3 : strlwr(string) – String LowerCase : This function is used to converts upper case letters of the string in to lower case letters.

Syntax : strlwr(string);

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  char str[]="JBREC";
  clrscr();
  strlwr(str);
  printf("The lowercase is :%s\n",str);
  getch();
}
```

OUTPUT : The lowercase is : jbrec

Write a program to which converts given string in to lowercase.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char str[10];
  int index;
  printf("Enter the string:");
  scanf("%s",str);
  for(index=0;str[index]!='\0';index++)
  {
    if(str[index]>='A' && str[index]<='Z')
            str[index]=str[index]+32;
  }
  printf("After conversionis :%s",str);
  getch();
}
```

OUTPUT : Enter the string : SUBBAREDDY

After conversion string is :subbareddy

4 : strupr(string) – String UpperCase : This function is used to converts lower case letters of the string in to upper case letters.

Syntax : strupr(string);

Program using strupr() function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  char str[]="jbrec";
  strupr(str);
  printf("UpperCase is :%s\n",str);
  getch();
}
```

OUTPUT : UpperCase is : JBREC

Write a program to which converts given string in to uppercase.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char str[10];
  int index;
  printf("Enter the string:");
  scanf("%s",str);
  for(index=0;str[index]!='\0';index++)
  {
```

```
   if(str[index]>='a' && str[index]<='z')
            str[index]=str[index]-32;
  }
  printf("After conversionis :%s",str);
  getch();
}
```

OUTPUT : Enter the string : subbareddy

After conversion string is :SUBBAREDDY

5 : strcmp(string1,string2) – String Comparision : This function is used to compares two strings to find out whether they are same or different. If two strings are compared character by character until the end of one of the string is reached. If the two strings are same strcmp() returns a value zero. If they are not equal, it returns the numeric difference between the first non-matching characters.

Syntax : strcmp(string1,string2);

Program using strcmp() function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  char str1[]="reddy";
  char str2[]="reddy";
  int i,j,k;
  i=strcmp(str1,str2);
  j=strcmp(str1,"subba");
  k=strcmp(str2,"Subba");
  printf("%5d%5d%5d\n",i,j,k);
```

```
}
```

OUTPUT : 0 -1 32

Write a C program to find the comparision of two strings.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  char str1[10],str2[20];
  int index,l1,l2,flag=1;
  printf("Enter first string:");
  scanf("%s",str1);
  printf("Enter second string:");
  scanf("%s",str2);
  l1=strlen(str1);
  l2=strlen(str2);
  printf("Length of string1:%d\n",l1);
  printf("Length of string2:%d\n",l2);
  if(l1==l2)
  {
    for(index=0;str1[index]!='\0';index++)
    {
      if(str1[index]!=str2[index])
      {
            flag=0;
```

```
        break;
      }
    }
  }
 else
   flag=0;
  if(flag==1)
  printf("Strings are equal");
  else
  printf("Strings are not equal");
}
```

OUTPUT : Enter the first string :jbrec

Enter the second string:jbrec

Length of string1 :5

Length of string2 :5

Strings are equal

6: strcat(string1,string2) – String Concatenation : This function is used to concatenate or combine, two strings together and forms a new concatenated string.

Syntax : strcat(sting1,string2);

Where

string1 : is the firdt string1.

string2 : is the second string2

when the above function is executed, string2 is combined with string1 and it removes the null character (\0) of string1 and places string2 from there.

Program using strcat() function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
 char str1[10]="jbrec";
 char str2[]="ece";
 strcat(str1,str2);
 printf("%s\n",str1);
 printf("%s\n",str2);
 getch();
}
```

OUTPUT : jbrecece

ece

7 : strrev(string) - String Reverse :This function is used to reverse a string. This function takes only one argument and return one argument.

Syntax : strrev(string);

Ex : Program using strrev() function

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

```
 char str[20];
 printf("Enter the string:");
 scanf("%s",str);
 printf("The string reversed is:%s",strrev(str));
 getch();
}
```

OUTPUT : Enter the string :subbareddy

The string reversed is : ydderabbus

UNIT – III

FUNCTIONS:

User-Defined Functions

Definition: A function is a block of code/group of statements/self contained block of statements/ basic building blocks in a program that performs a particular task. It is also known as ***procedure*** or ***subroutine*** or **module**, in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides *modularity* and code *reusability*.

Advantage of functions

1) **Code Reusability**

By creating functions in C, you can call it many times. So we don't need to write the same code again and again.

2) **Code optimization**

It makes the code optimized we don't need to write much code.

3) **Easily to debug the program.**

Example: Suppose, you have to check 3 numbers (781, 883 and 531) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

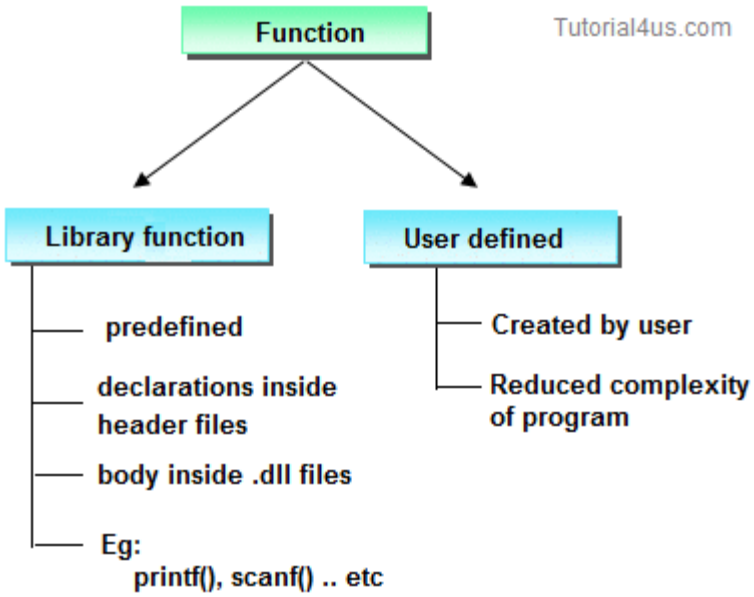
1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc. You just need to include appropriate header files to use these functions. **These are already declared and defined in C libraries. oints to be Remembered**

   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.

2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code. Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.



ELEMENTS OF USER-DEFINED FUNCTINS :

In order to write an efficient user defined function, the programmer must familiar with the following three elements.

1 : Function Declaration. (Function Prototype).

2 : Function Call.

3 : Function Definition

Function Declaration. (Function Prototype).

A function declaration is the process of tells the compiler about a function name.

Syntax

return_type function_name(parameter/argument);

return_type function-name();

Ex : int add(int a,int b);

int add();

Note: At the time of function declaration function must be terminated with **;**.

Calling a function/function call

When we call any function control goes to function body and execute entire code.

Syntax : function-name();

function-name(parameter/argument);

return value/ variable = function-name(parameter/argument);

Ex : add(); // function without parameter/argument

add(a,b); // function with parameter/argument

c=fun(a,b); // function with parameter/argument and return values

Defining a function.

Defining of function is nothing but give body of function that means write logic inside function body.

Syntax

return_ type function-name(parameter list) **// function header.**

{

declaration of variables;

body of function; **// Function body**

return statement; (expression or value) //**optional**

}

Eg:

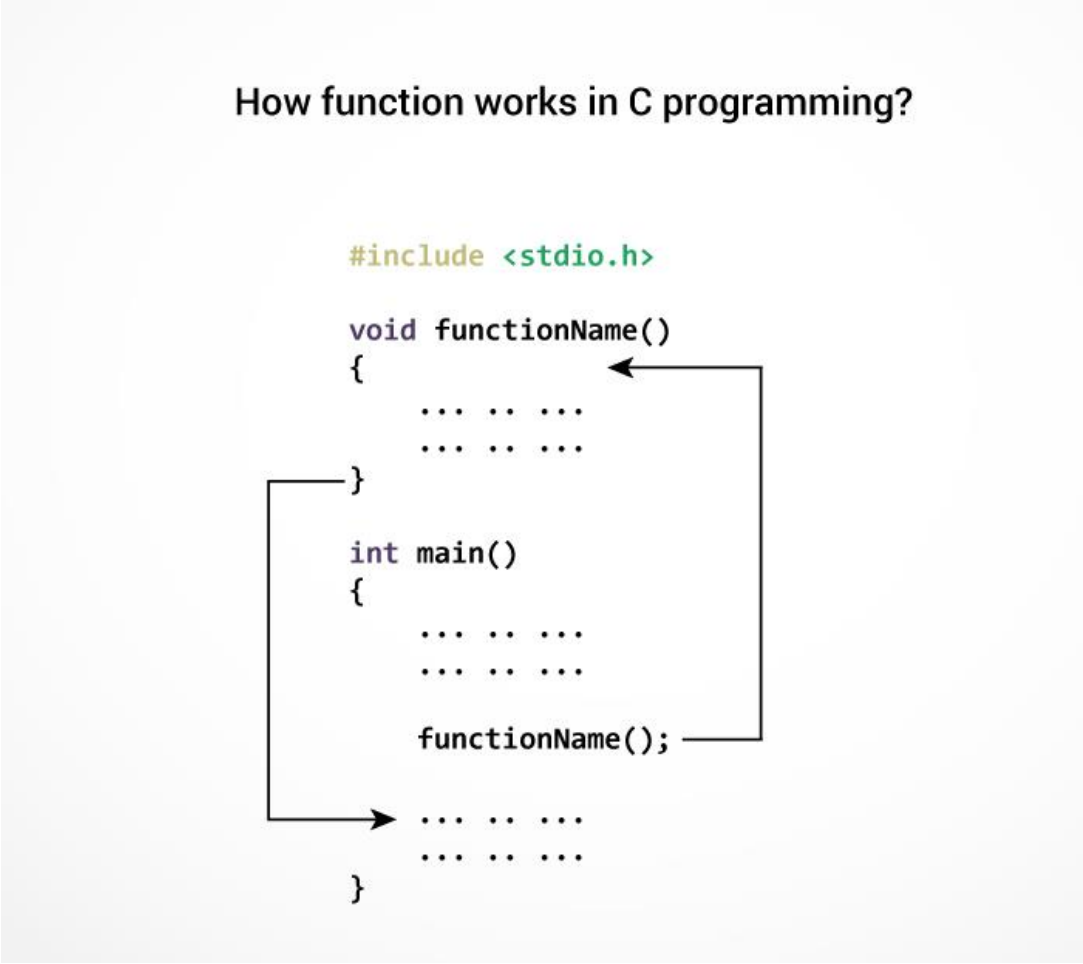
```
int add( int x, int y)
{
    int z;
    z = x + y;
}
```

(**or**)

```
int add( int x, int y)
{
return ( x + y );
}
```

return z;

}



The execution of a C program begins from the main() function.

When the compiler encounters functionName(); inside the main function, control of the program jumps to

void functionName()

And, the compiler starts executing the codes inside the user-defined function.

The control of the program jumps to statement next to functionName(); once all the codes inside the function definition are executed.

Example:

#include<stdio.h>

```
#include<conio.h>

void sum(); // declaring a function
clrsct();
int a=10,b=20, c;
void sum()  // defining function
{
c=a+b;
printf("Sum: %d", c);
}
void main()
{
sum();  // calling function
}
```

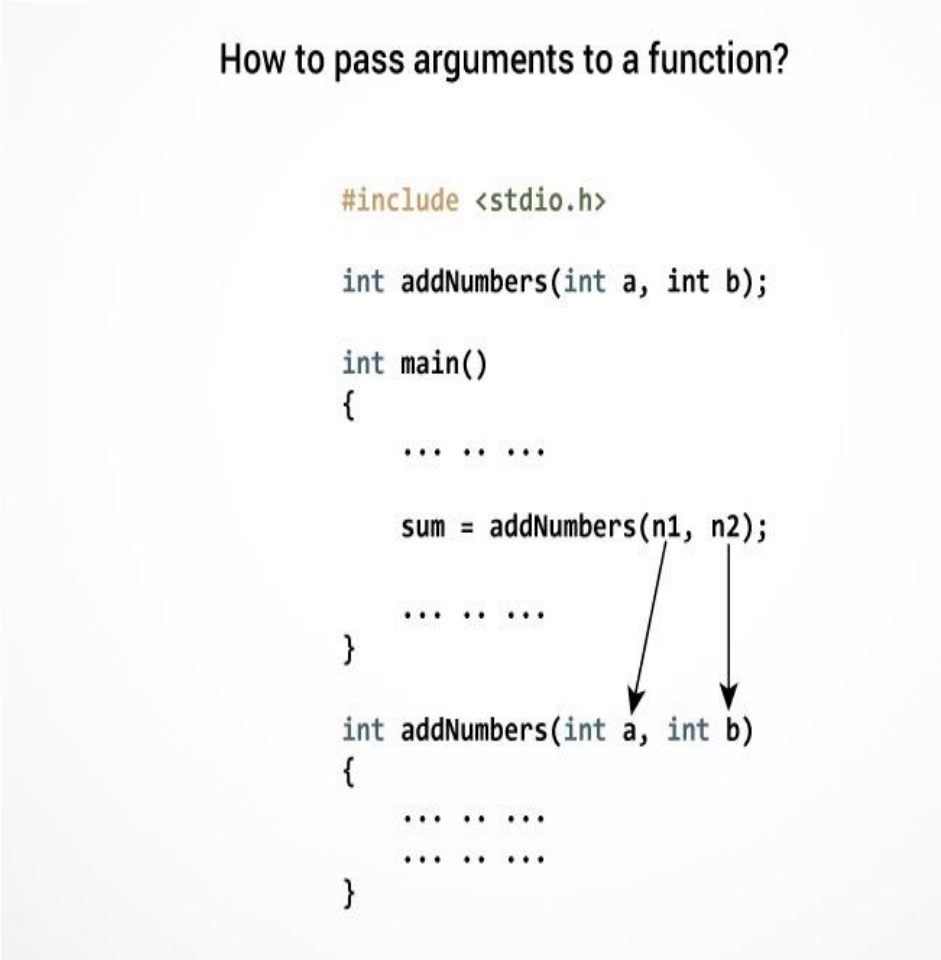
Output

Sum:30

Example:

```
#include <stdio.h>
int addNumbers(int a, int b);         // function prototype
int main()
{
   int n1,n2,sum;
   printf("Enters two numbers: ");
   scanf("%d %d",&n1,&n2);
   sum = addNumbers(n1, n2);         // function call
```

```
    printf("sum = %d",sum);
    return 0;
}
int addNumbers(int a,int b)         // function definition
{
    int result;
    result = a+b;
    return result;                  // return statement
}
```



Return Statement

Syntax of return statement

Syntax : return; // **does not return any value**

or

return(exp); // **the specified exp value to calling function.**

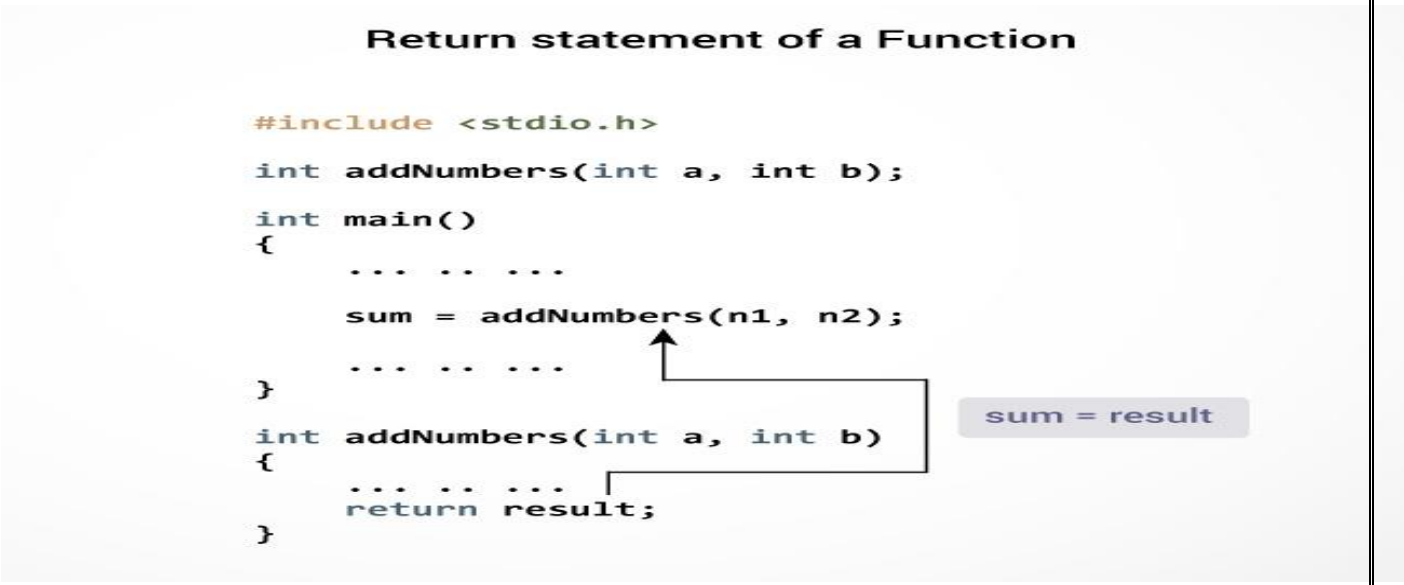
For example,

return a;

return (a+b);

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

In the above example, the value of variable result is returned to the variable sum in the main() function.



PARAMETERS :

parameters provides the data communication between the calling function and called function.

They are two types of parametes

1 : Actual parameters.

2 : Formal parameters.

1 : Actual Parameters : These are the parameters transferred from the calling function (main program) to the called function (function).

2 : Formal Parameters : These are the parameters transferred into the calling function (main program) from the called function(function).

- The parameters specified in calling function are said to be Actual Parameters.
- The parameters declared in called function are said to be Formal Parameters.
- The value of actual parameters is always copied into formal parameters.

Ex :

```
main()
{
  fun1( a , b );   //Calling function
}
fun1( x, y )      //called function
{
   . . . . . .
}
```

Where

a, b are the **Actual Parameters**

x, y are the **Formal Parameters**

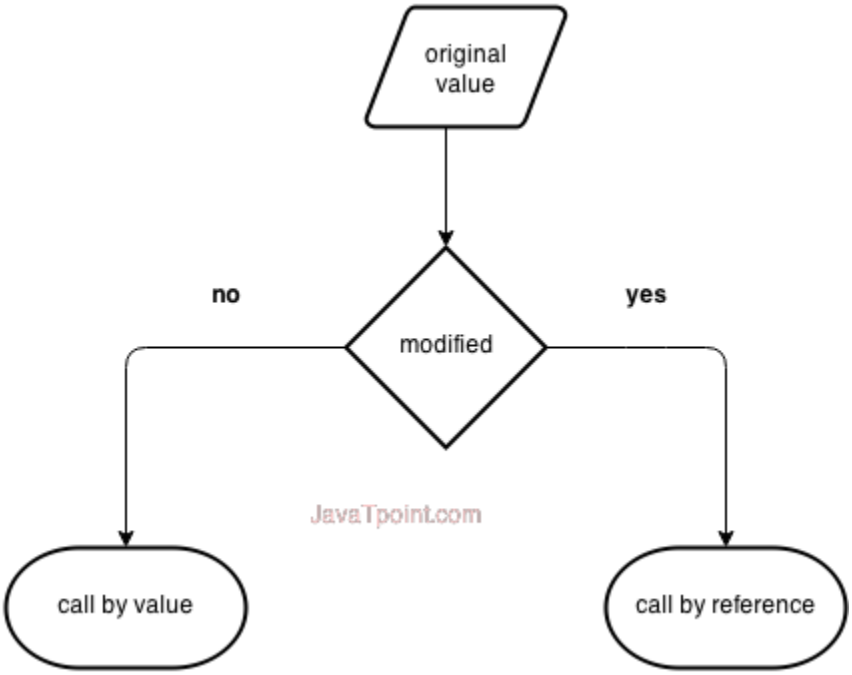
Difference between Actual Parameters and Formal Parameters

Actual Parameters	**Formal Parameters**
1 : Actual parameters are used in calling function when a function is invoked.	**1 :** Formal parameters are used in the function header of a called function.

Ex : c=add(a,b); Here a,b are actual parameters.	**Ex :** int add(int m,int n); Here m,n are called formal parameters.
2 : Actual parameters can be constants, variables or expression. **Ex :** c=add(a,b) **//variable** c=add(a+5,b); **//expression.** c=add(10,20); **//constants.**	**2 :** Formal parametes should be only variable. Expression and constants are not allowed. **Ex :** int add(int m,n); **//CORRECT** int add(int m+n,int n) **//WRONG** int add(int m,10); **//WRONG**
3 : Actual parameters sends values to the formal parameters. **Ex :** c=add(4,5);	**3 :** Formal parametes receive values from the actual parametes. **Ex :** int add(int m,int n); Here m will have the value 4 and n will have the value 5.
4 : Address of actual parameters can be sent to formal parameters	**4 :** if formal parameters contains address, they should be declared as pointers.

PASSING PARAMETERS TO FUNCTIONS : There are two ways to pass value or data to function in C language: ***call by value*** and ***call by reference*****.** Original value is **not modified in**

call by value but it is **modified in call by reference.**



The called function receives the information from the calling function through the parameters. The variables used while invoking the calling function are called actual parameters and the variables used in the function header of the called function are called formal parameters.

C provides two mechanisms to pass parameters to a function.

1 : Pass by value (OR) Call by value.

2 : Pass by reference (OR) Call by Reference.

1 : Pass by value (OR) Call by value :

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Or

When a function is called with actual parameters, the values of actual parameters are copied into formal parameters. If the values of the formal parametes changes in the function, the values of the actual parameters are not changed. This way of passing parameters is called pass by value or call by value.

Ex :

```
#include<stdio.h>
#include<conio.h>
void swap(int ,int );
void main()
{
   int i,j;
   printf("Enter i and j values:");
   scanf("%d%d",&i,&j);
   printf("Before swapping:%d%d\n",i,j);
   swap(i,j);
   printf("After swapping:%d%d\n",i,j);
   getch();
 }
void swap(int a,int b)
{
   int temp;
   temp=a;
   a=b;
   b=temp;
}
```

Output

Enter i and j values: 10 20

Before swapping: 10 20

After swapping: 10 20

2 : Pass by reference (OR) Call by Reference : In pass by reference, a function is called with addresses of actual parameters. In the function header, the formal parameters receive the addresses of actual parameters. Now the formal parameters do not contain values, instead they contain addresses. Any variable if it contains an address, it is called a pointer variable. Using pointer variables, the values of the actual parameters can be changed. This way of passing parameters is called call by reference or pass by reference.

Ex :

```
#include<stdio.h>
#include<conio.h>
void swap(int *,int *);
void main()
{
   int i,j;
   printf("Enter i and j values:");
   scanf("%d%d",&i,&j);
   printf("Before swapping:%d%d\n",i,j);
   swap(&i ,&j);
  printf("After swapping:%d%d\n",i,j);
 }
void swap(int *a,int *b)
{
   int temp;
   temp=*a;
   *a=*b;
   *b=temp;
```

}

Output

Enter i and j values: 10 20

Before swapping:10 20

After swapping: 20 10

Differnce between Call by value and Call by reference

Call by value	**Call by Reference**
1 : When a function is called the values of variables are passed	1 : When a function is called the address of variables are passed.
2 : Change of formal parameters in the function will not affect the actual parameters in the calling function.	**2 :** The actual parameters are changed since the formal parameters indirectly manipulate the actual parametes.
3 : Execution is slower since all the values have to be copied into formal parameters.	**3 :** Execution is faster since only address are copied.

1 : Functions with no Parameters and no Return Values.

2 : Functions with no Parameters and Return Values.

3 : Functions with Parameters and no Return Values.

4 : Functions with Parameters and Return Values.

1 : Functions with no Parameters and no Return Values :

1 : In this category, there is no data transfer between the calling function and called function.

2 : But there is flow of control from calling function to the called function.

3 : When no parameters are there , the function cannot receive any value from the calling function.

4: When the function does not return a value, the calling function cannot receive any value from the called function.

Ex #include<stdio.h>

```
#include<conio.h>
void sum();
void main()
{
   sum();
   getch();
}
void sum()
{
  int a,b,c;
  printf("enter the values of a and b");
  scanf("%d%d",&a,&b);
  c=a+b;
  printf("sum=%d",c);
}
```

2 : Functions with no Parameters and Return Values.

1 : In this category, there is no data transfer between the calling function and called function.

2 : But there is data transfer from called function to the calling function.

3 : When no parameters are there , the function cannot receive any values from the calling function.

4: When the function returns a value, the calling function receives one value from the called function.

Ex :

```
#include<stdio.h>
#include<conio.h>
int sum();
```

```
 void main()
{
   int c;
   clrscr();
   c=sum();
   printf("sum=%d",c);
   getch();
 }
 int sum()
{
  int a,b,c;
  printf("enter the values of a and b");
  scanf("%d%d",&a,&b);
  c=a+b;
  return c;
}
```

3 : Functions with Parameters and no Return Values.

1 : In this category, there is data transfer from the calling function to the called function using parameters.

2 : But there is no data transfer from called function to the calling function.

3 : When parameters are there , the function can receive any values from the calling function.

4: When the function does not return a value, the calling function cannot receive any value from the called function.

Ex :

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

```
 void sum(int a,int b);
 void main()
{
    int m,n;
    clrscr();
    printf("Enter m and n values:");
    scanf("%d%d",&m,&n);
    sum(m,n);
    getch();
}
void sum(int a,int b)
{
   int c;
  c=a+b;
  printf("sum=%d",c);
}
```

4 : Functions with Parameters and Return Values.

1 : In this category, there is data transfer from the calling function to the called function using parameters.

2 : But there is no data transfer from called function to the calling function.

3 : When parameters are there , the function can receive any values from the calling function.

4: When the function returns a value, the calling function receive a value from the called function.

Ex :

```
#include<stdio.h>
#include<conio.h>
int sum(int a,int b);
```

```
void main()
{
    int m,n,c;
    clrscr();
    printf("Enter m and n values");
    scanf("%d%d",&m,&n);
    c=sum(m,n);
    printf("sum=%d",c);
    getch();
}
int sum(int a,int b)
{
    int c;
    c=a+b;
    return c;
}
```

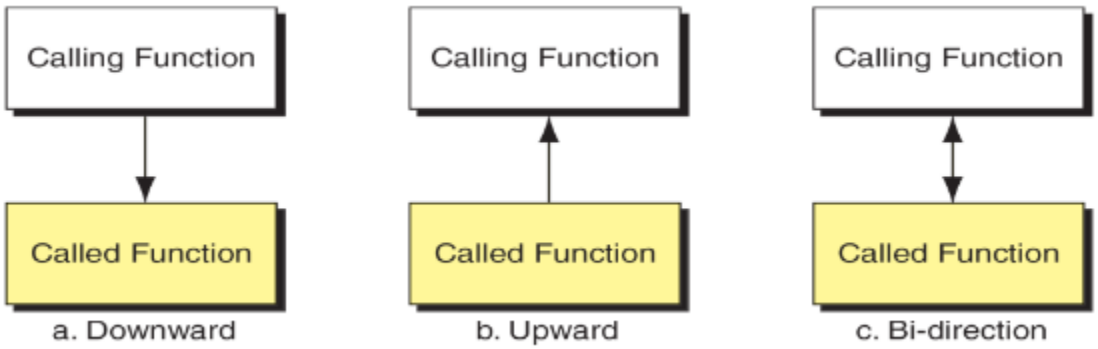
Inter-Function Communication

When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function. In this process, both calling and called functions have to communicate each other to exchange information. The process of exchanging information between calling and called functions is called as inter function communication.

In C, the inter function communication is classified as follows...

- Downward Communication
- Upward Communication

- Bi-directional Communication



Downward Communication

In this type of inter function communication, **the data is transferred from calling function to called function** but not from called function to calling function. The functions with parameters and without return value are considered under downward communication. In the case of downward communication, the execution control jumps from calling function to called function along with parameters and executes the function definition,and finally comes back to the calling function without any return value. For example consider the following program...

Example:

```
#include <stdio.h>

#include<conio.h>

void main(){

  int num1, num2 ;

  void addition(int, int) ; // function declaration

  clrscr() ;

  num1 = 10 ;

  num2 = 20 ;

  printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;

  addition(num1, num2) ; // calling function

    getch() ;
```

```
}
void addition(int a, int b)  // called function
{
   printf("SUM = %d", a+b) ;
 }
```

Output

SUM=30

Upward Communication

In this type of inter function communication, the **data is transferred from called function to calling function** but not from calling function to called function. The functions without parameters and with return value are considered under upward communication. In the case of upward communication, the execution control jumps from calling function to called function without parameters and executes the function definition, and finally comes back to the calling function along with a return value. For example consider the following program...

Exmaple:

```
#include <stdio.h>
#include<conio.h>
void main(){
   int result ;
   int addition() ; // function declaration
   clrscr() ;
   result = addition() ; // calling function
   printf("SUM = %d", result) ;
   getch() ;
}
int addition()  // called function
```

```
{
  int num1, num2 ;
  num1 = 10;
  num2 = 20;
  return (num1+num2) ;
}
```

Output

SUM=30

Bi - Directional Communication

In this type of inter function communication, the data is transferred from calling function to called function and also from called function to calling function. The functions with parameters and with return value are considered under bi-directional communication. In the case of bi-drectional communication, the execution control jumps from calling function to called function along with parameters and executes the function definition, and finally comes back to the calling function along with a return value. For example consider the following program...

Example:

```
#include <stdio.h>
#include<conio.h>
void main(){
  int num1, num2, result ;
  int addition(int, int) ; // function declaration
  clrscr() ;
  num1 = 10 ;
  num2 = 20 ;
  result = addition(num1, num2) ; // calling function
  printf("SUM = %d", result) ;
```

```
  getch() ;
}
int addition(int a, int b)  // called function
{
  return (a+b) ;
}
```

Output

SUM=30

Standard Functions

The standard functions are built-in functions. In C programming language, the standard functions are declared in header files and defined in .dll files. In simple words, the standard functions can be defined as "the ready made functions defined by the system to make coding more easy". The standard functions are also called as library functions or pre-defined functions.

In C when we use standard functions, we must include the respective header file using #include statement. For example, the function printf() is defined in header file stdio.h (Standard Input Output header file). When we use printf() in our program, we must include stdio.h header file using #include<stdio.h> statement.

C Programming Language provides the following header files with standard functions.

Header File	Purpose	Example Functions
stdio.h	Provides functions to perform standard I/O operations	printf(), scanf()
conio.h	Provides functions to perform console I/O operations	clrscr(), getch()
math.h	Provides functions to perform mathematical operations	sqrt(), pow()
string.h	Provides functions to handle string data values	strlen(), strcpy()
stdlib.h	Provides functions to perform general functions	calloc(), malloc()

time.h	Provides functions to perform operations on time and date	time(), localtime()
ctype.h	Provides functions to perform - testing and mapping of character data values	isalpha(), islower()
setjmp.h	Provides functions that are used in function calls	setjump(), longjump()
signal.h	Provides functions to handle signals during program execution	signal(), raise()
assert.h	Provides Macro that is used to verify assumptions made by the program	assert()
locale.h	Defines the location specific settings such as date formats and currency symbols	setlocale()
stdarg.h	Used to get the arguments in a function if the arguments are not specified by the function	va_start(), va_end(), va_arg()
errno.h	Provides macros to handle the system calls	Error, errno
float.h	Provides constants related to floating point data values	
limits.h	Defines the maximum and minimum values of various variable types like char, int and long	
stddef.h	Defines various variable types	
graphics.h	Provides functions to draw graphics.	circle(), rectangle()

STANDARD 'C' LIBRARY FUNCTIONS

1 : stdio.h

2 : stdlib.h

3 : string.h

4 : math.h

5 : ctype.h

6 : time.h

1 : STANDARD I/O LIBRARY FUNCTIONS <STDIO.H>

Functions	**DataType**	**Purpose**
printf()	int	Send data items to the standared output device.
scanf()	int	Enter data items from the standard input device.
gets(s)	char	Enter string s from the standard input device.
getc(f)	int	Enter a string character from file f.
getchar()	int	Enter a single character from the standard input device.
putc(c,f)	int	Send a single character to file f.
puts(s)	int	Send string s to the standard output device.
putchar(c)	int	Send a single character to the standard output device.
fgetc(f)	int	Enter a single character from file f.
fgets(s,I,f)	char	Enter string s, containing I characters, from file f.
fprintf(f)	int	Send data items to file f.
fscanf(f)	int	Enter data items from file f.
fputc(c,f)	int	Send a single character to file f.
fputs(s,f)	int	Send string s to file f.
fread(s,il,i2,f)	int	Enter i2 data items, each of size i1 bytes, from file f.
fclose(f)	int	Close file f, return 0 if file is successfully closed.

2 : STANDARD LIBRARY FUNCTIONS <STDLIB.H>

Functions	DataType	Purpose
abs(i)	int	Return the absolute value of i.
atof(s)	double	Convert string s to a double-precesion quantity.
calloc(u1,u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes. Return a pointer to the beginning of the allocated space.
exit(u)	void	Close all files and buffers, and terminate the program.
free(p)	void	Free a block of allocated memory whose beginning is indicated by p.
malloc(u)	void*	Allocate u bytes of memory.
rand()	int	Return a random positive integer.
realloc(p,u)	void*	Allocate u bytes of new memory to the pointer variable p, return a pointer to the beginning of the new memory space.
system(s)	int	Pass command string s to the operating system.
srand(u)	void	Initialize the random number generator.

3 : STRING LIBRARY FUNCTIONS <STRING.H>

Functions	DataType	Purpose
strlen()		Finds length of string
strlwr()		Converts a string to lowercase
strupr()		Converts a string to uppercase
strcat()		Appends one string at the end of another
strcpy()		Copies a string into another
strcmp()		Compares two strings
strrev()		Reverses string

4 : MATH LIBRARY FUNCTIONS <MATH.H>

Functions	DataType	Purpose
acos(d)	double	Return the arc cosine of d.
atan(d)	double	Return the arc tangent of d.
asin(d)	double	Return the arc sine of d.
ceil(d)	double	Return a value rounded up to the next higher integer.
cos(d)	double	Return the cosine of d.
cosh(d)	double	Return the hyperbolic cosine of d.
exp(d)	double	Raise e to the power d.
fabs(d)	double	Return the absolute value of d.
floor(d)	double	Return a value rounded down to the next lower integer.
labs(l)	long int	Return the absolute value of l.
log(d)	double	Return the natural logarithm of d.
pow(d1,d2)	double	Return d1 raised to the d2 power.
sin(d)	double	Return the sine of d.
sqrt(d)	double	Return squre root of d.
tan(d)	double	Return the tangent of d.

5 : CHARACTER LIBRARY FUNCTIONS <CTYPE.H>

Functions	DataType	Purpose

isalnum(c)	Int	Determine if argument is alphanumeric. Return nonzero value if true, 0 otherwise.
isalpha(c)	Int	Determine if argument is alphabetic. Return nonzero value if true, 0 otherwise.
isascii(c)	Int	Determine if argument is an ASCII character,. Return nonzero value if true, 0 otherwise.
isdigit(c)	Int	Determine if argument is a decimal digit. Return nonzero value if true, 0 otherwise.
isgraph(c)	Int	Determine if argument is a graphic printing ASCII Character. Return nonzero value if true, 0 otherwise.
islower(c)	Int	Determine if argument is lowercase. Return nonzero value if true, 0 otherwise.
isprint(c)	Int	Determine if argument is a printing ASCII character. Return nonzero value if true, 0 otherwise.
isspace(c)	Int	Determine if argument is a whitespace character. Return nonzero value if true, 0 otherwise.
isupper(c)	Int	Determine if argument is uppercase. Return nonzero value if true, 0 otherwise.
toascii(c)	Int	Convert value of argument to ASCII
tolower(c)	Int	Convert letter to lowercase
toupper(c)	Int	Convert letter to uppercase.

6 : TIME LIBRARY FUNCTIONS <TIME.H>

Functions	**DataType**	**Purpose**
difftime(11,12)	double	Return the time difference 11-12, where 11 and 12 represent elapsed time beyond a designated base time.
time(p)	long int	Return the number of seconds elapsed beyond a designated base time.

Storage Classes

In C language, each variable has a storage class which is used to define scope and life time of a variable.

Storage: Any variable declared in a program can be stored either in memory or registers. Registers are small amount of storage in CPU. The data stored in registers has fast access compared to data stored in memory.

Storage class of a variable gives information about the location of the variable in which it is stored, initial value of the variable, if storage class is not specified; scope of the variable; life of the variable.

There are four storage classes in C programming.

1 : Automatic Storage class.

2 : Register Storage class.

3 : Static Storage class.

4 : External Storage class.

1: Automatic Storage class : To define a variable as automatic storage class, the keyword 'auto' is used. By defining a variable as automatic storage class, it is stored in the memory. The default value of the variable will be garbage value. Scope of the variable is within the block where it is defined and the life of the variable is until the control remains within the block.

Syntax : auto data_type variable_name;

auto int a,b;

Example:

```
void main()
{
 int detail;
 or
 auto int detail;    //Both are same
}
```

The variables a and b are declared as integer type and auto. The **keyword auto is not mandatory.** Because the **default storage class in C is auto.**

Note: A variable declared inside a function without any storage class specification, is **by default an automatic variable. Automatic variables can also be called local variables because they are local to a function.**

Ex :

```
void function1();
void function2();
void main()
{
    int x=100;
    function2();
    printf("%d",x);
 }
 void function1()
 {
   int x=10;
   printf("%d",x);
 }
 void function2()
{
   int x=0;
   function1();
  printf("%d",x);
}
```

OUTPUT

10

0

100

2: Register Storage class : To define a variable as register storage class, the **keyword 'register' is used.** If CPU cannot store the variables in CPU registers, then the variables are

assumed as auto and stored in the memory. **When a variable is declared as register, it is stored in the CPU registers.** The **default value of the variable will be garbage value**. Scope of the variable within the block where it is defined and the life of the variables is until the control remains within the block.

Register variable has faster access than normal variable. Frequently used variables are kept in register. Only few variables can be placed inside register.

NOTE : We can't get the address of register variable

Sytax : register data_type variable_name;

Ex: register int i;

Ex :

```
void demo();
void main()
{
    demo();
    demo();
    demo();
}
void demo()
{
    register int i=20;
   printf("%d\n",i);
   i++;
}
```

OUTPUT

20

20

20

3 : Static Storage class : When a variable is declared as static, it is stored in the memory. The **default value of the variable will be zero**. Scope of the variable is within the block where it is defined and the life of the variable persists between different function calls. To define a variable as static storage class, **the keyword 'static' is used**. A static variable can be initialized only once, it cannot be reinitialized.

Syntax : static data_type variable_name;

Ex: static int i;

Ex :

```
void demo();
void main()
{
    demo();
    demo();
    demo();
}
void demo()
{
    static  int i=20;
    printf("%d",i);
    i++;
}
```

OUTPUT

20

21

22

4 : External Storage class : When a **variable is declared as extern**, it is stored in the memory. The **default value is initialized to zero**. The scope of the variable is global and the life of the variable is until the program execution comes to an end. To define a variable as external storage class, the **keyword 'extern'** is used. An extern variable is also called as a global variable. **Global** variables remain available throughout the entire program. One important thing to remember about global variable is that their values can be changed by any function in the program.

Systax : extern data_type variable_name;

extern int i;

Ex:

```
int number;
void main()
{
 number=10;
}
fun1()
{
 number=20;
}
fun2()
{
 number=30;
}
```

Here the global variable **number** is available to all three functions.

Ex :

```
void fun1();
    void fun2();
    int e=20;
    void main()
   {
      fun1();
     fun2();
   }
  void fun1()
```

```
        {
          extern int e;
          printf("e number is :%d",e);
        }
        void fun2()
      {
          printf("e number is :%d",e);
    }
```

extern keyword

The extern keyword is used before a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.

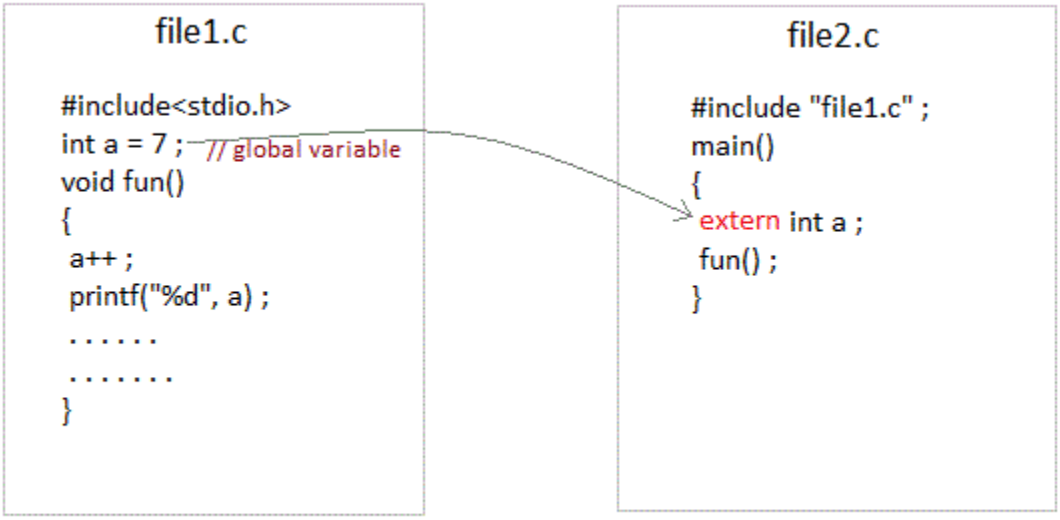
Problem when extern is not used

```
main()
{
  a = 10;      //Error:cannot find variable a
  printf("%d",a);
}
```

Example Using extern in same file

```
main()
{
  extern int x;  //Tells compiler that it is defined somewhere else
  x = 10;
  printf("%d",x);
}
```

int x; *//Global variable x*



global variable from one file can be used in other using **extern** keyword.

Storage Classes	Storage Place	Default Value	Scope	Life-time
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of main program, May be declared anywhere in the program
static	RAM	Zero	Local	Till the end of main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within function

Recursion

When *function is called within the same function*, it is known as recursion in C. The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is know as tail recursion. In tail recursion, we generally call the same function with return statement.

Features :

- There should be at least one if statement used to terminate recursion.
- It does not contain any looping statements.

Advantages :

- It is easy to use.
- It represents compact programming structures.

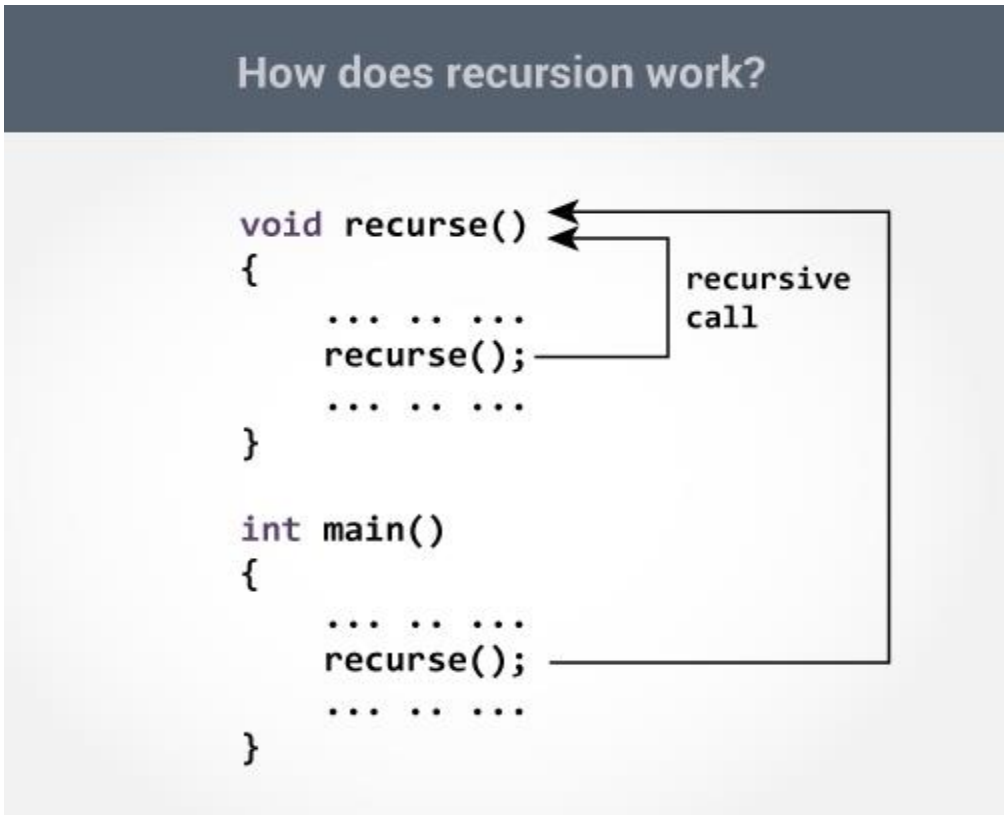
Disadvantages :

- It is slower than that of looping statements because each time function is called.

Note: while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example of recursion.

```
recursionfunction(){
recursionfunction();//calling self function
}
```



Example of tail recursion in C

// print factorial number using tail recursion

```
#include<stdio.h>
#include<conio.h>
int factorial (int n)
{
   if ( n < 0)
      return -1; /*Wrong value*/
   if (n == 0)
      return 1; /*Terminating condition*/
   return (n * factorial (n -1));
}
 void main(){
```

```
int fact=0;
clrscr();
fact=factorial(5);
printf("\n factorial of 5 is %d",fact);
 getch();  }
```
Outputfactorial of 5 is 120

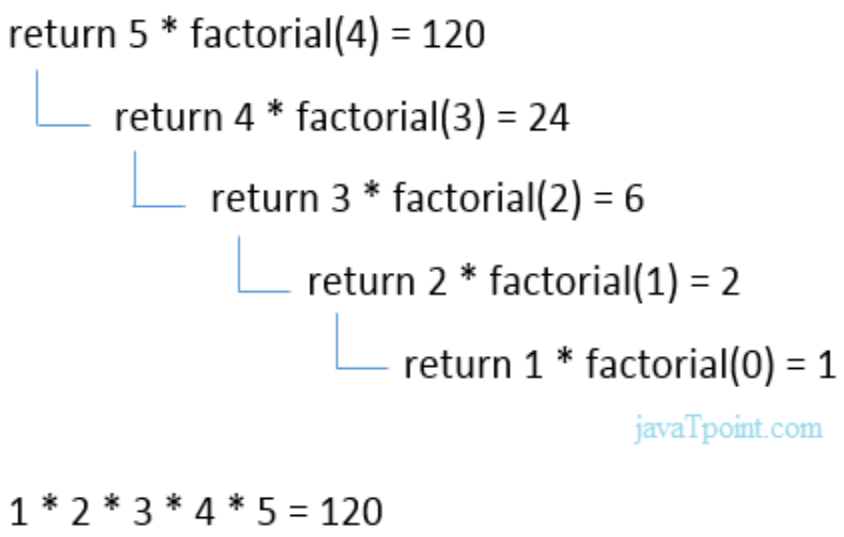
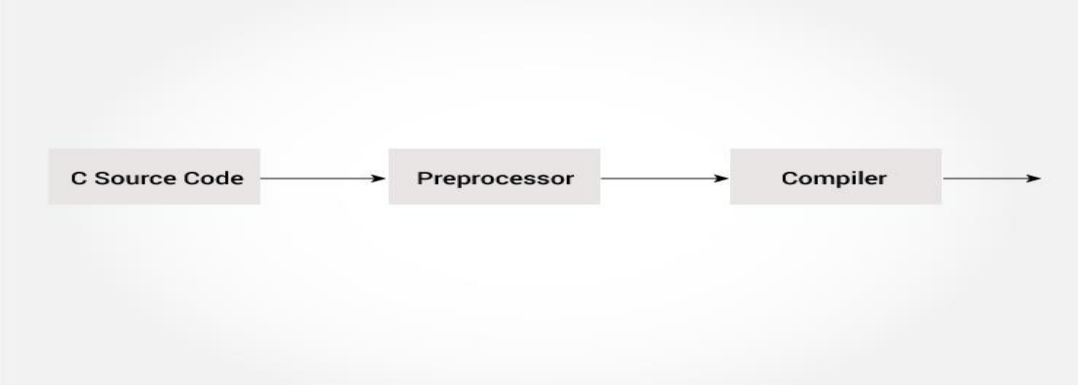
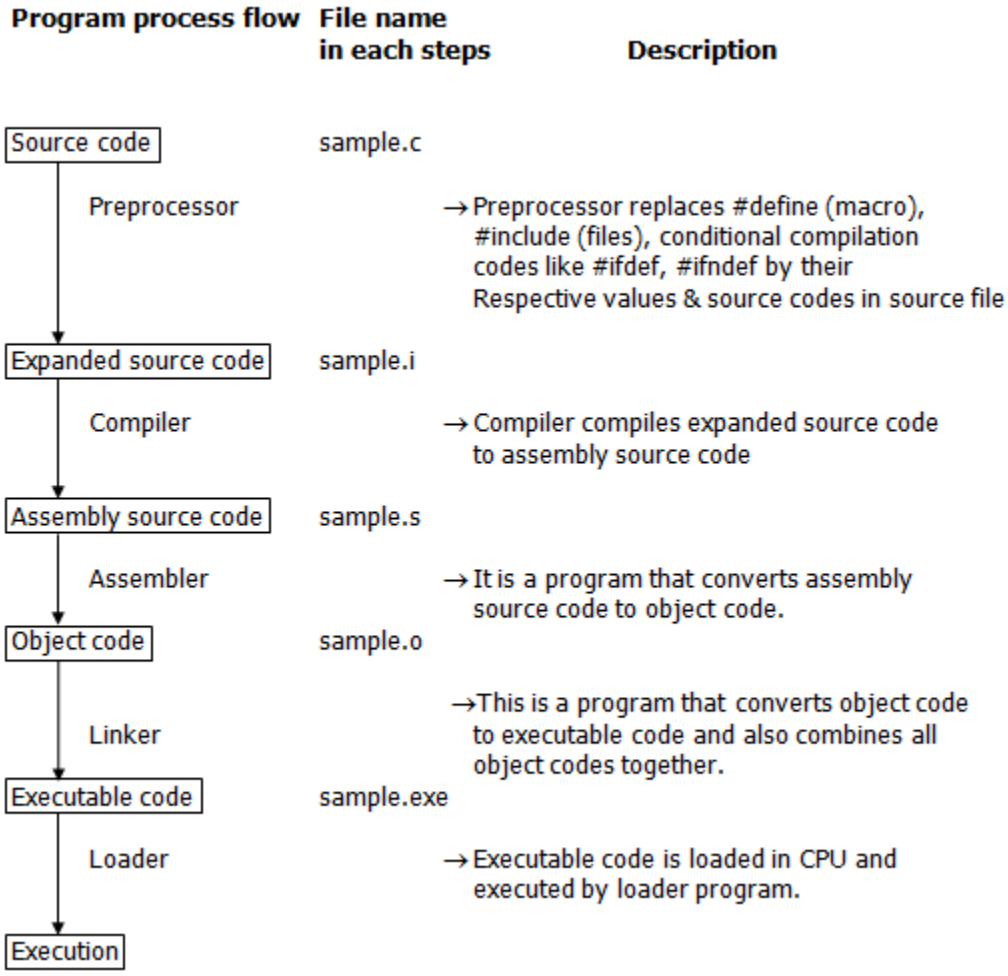


Fig: Recursion

Preprocessor Commands



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.



The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the

compiler to do required pre-processing before the actual compilation. All preprocessor commands begin with a hash symbol (#).

list of preprocessor directives.

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma

C Macros

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

```
#define PI 3.14
```

Here, PI is the macro name which will be replaced by the value 3.14.

Function-like Macros

The function-like macro looks like function call. For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Here, MIN is the macro name.

Predefined Macros

There are some predefined macros which are readily for use in C programming.

No.	Macro	Description
1	_DATE_	represents current date in "MMM DD YYYY" format.
2	_TIME_	represents current time in "HH:MM:SS" format.
3	_FILE_	represents current file name.
4	_LINE_	represents current line number.
5	_STDC_	It is defined as 1 when compiler complies with the ANSI standard.

```
#include <stdio.h>
main() {
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );
}
```

Output

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
```

Line :8

ANSI :1

C #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files. There are two variants to use #include directive.

1. #include <filename>
2. #include "filename"

The **#include <filename>** tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

The **#include "filename"** tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

```
1. #include <stdio.h>
2. main() {
3.     printf("Hello C");
4. }
```

Output:

Hello C

#include notes:

Note 1: In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

Note 2: In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

Note 3: You can use only comment after filename otherwise it will give error.

C #define

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

#define token value

Example of #define to define a constant.

```
#include <stdio.h>

#define PI 3.14

 main() {

   printf("%f",PI);

}
```

Output:

3.140000

Example of #define to create a macro.

```
#include <stdio.h>

#define MIN(a,b) ((a)<(b)?(a):(b))

void main() {

   printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));

}
```

Output:

Minimum between 10 and 20 is: 10

C #undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:

```
#undef token
```

Simple example to define and undefine a constant.

```
#include <stdio.h>

#define PI 3.14

#undef PI

main() {

  printf("%f",PI);

}
```

Output:

Compile Time Error: 'PI' undeclared

The #undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variable. But before being undefined, it was used by square variable.

```
#include <stdio.h>

#define number 15

int square=number*number;

#undef number

main() {

  printf("%d",square);

}
```

Output:

225

C #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifdef MACRO
//code
#endif
```

Syntax with #else:

```
#ifdef MACRO
//successful code
#else
//else code
#endif
```

C #ifdef example

```
#include <stdio.h>
#include <conio.h>
#define NOINPUT
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
```

```
printf("Value of a: %d\n", a);
getch();
}
```

Output:

Value of a: 2

if you don't define NOINPUT, it will ask user to enter a number.

```
#include <stdio.h>
#include <conio.h>
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
  printf("Value of a: %d\n", a);
getch();
}
```

Output:

Enter a:5

Value of a: 5

C #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
//code
#endif
```

Syntax with #else:

```
#ifndef MACRO
//successful code
#else
//else code
#endif
```

C #ifndef example

simple example to use #ifndef preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define INPUT
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
```

```
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Enter a:5
Value of a: 5
```

if you don't define INPUT, it will execute the code of #ifndef.

```
#include <stdio.h>
#include <conio.h>
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Value of a: 2
```

C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

```
#if expression
//code
#endif
```

Syntax with #else:

```
#if expression
//if code
#else
//else code
#endif
```

Syntax with #elif and #else:

```
#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif
```

C #if example

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 0
```

```
void main() {
#if (NUMBER==0)
printf("Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output:

Value of Number is: 0

Another example to understand the #if directive clearly.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main() {
clrscr();
#if (NUMBER==0)
printf("1 Value of Number is: %d",NUMBER);
#endif
 #if (NUMBER==1)
printf("2 Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output:

2 Value of Number is: 1

C #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
#if expression
//if code
#else
//else code
#endif
```

Syntax with #elif:

```
#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif
```

C #else example

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main() {
#if NUMBER==0
printf("Value of Number is: %d",NUMBER);
#else
```

```
print("Value of Number is non-zero");
#endif
getch();
}
```

Output:

Value of Number is non-zero

C #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

Output:

Compile Time Error: First include then compile

if you include math.h, it does not gives error.

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

```
#ifndef __MATH_H
#error First include then compile
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

Output:

2.645751

C #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.

Syntax:

#pragma token

Different compilers can provide different usage of #pragma directive.

The turbo C++ compiler supports following #pragma directives.

#pragma argsused

#pragma exit

#pragma hdrfile

#pragma hdrstop

#pragma inline

#pragma option

#pragma saveregs

```
#pragma startup

#pragma warn
```

Example to use #pragma preprocessor directive.

```
#include<stdio.h>

#include<conio.h>

void func() ;

#pragma startup func

#pragma exit func

void main(){

printf("\nI am in main");

getch();

}

void func(){

printf("\nI am in func");

getch();

}
```

Output:

```
I am in func

I am in main

I am in func
```

KEY POINTS TO REMEMBER:

1. Source program is converted into executable code through different processes like precompilation, compilation, assembling and linking.
2. Local variables uses stack memory.
3. Dynamic memory allocation functions use the heap memory.

preprocessor	Syntax/Description
Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file "file_name" is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	**Syntax:** #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.

DIFFERENCE BETWEEN STACK & HEAP MEMORY IN C LANGUAGE?

Stack	Heap
Stack is a memory region where "local variables", "return addresses of function calls" and "arguments to functions" are hold while C program is executed.	Heap is a memory region which is used by dynamic memory allocation functions at run time.

CPU's current state is saved in stack memory	Linked list is an example which uses heap memory.

DIFFERENCE BETWEEN COMPILERS VS INTERPRETERS IN C LANGUAGE?

Compilers	**Interpreters**
Compiler reads the entire source code of the program and converts it into binary code. This process is called compilation. Binary code is also referred as machine code, executable, and object code.	Interpreter reads the program source code one line at a time and executing that line. This process is called interpretation.
Program speed is fast.	Program speed is slow.
One time execution. Example: C, C++	Interpretation occurs at every line of the program. Example: BASIC

The following section lists down all the important preprocessor directives −

Directive	**Description**
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.

#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

There are three types of preprocessor commands.

1 : macro substitution.

2 : file inclusion.

3 : conditional compilation directives.

1: Macro Substitution : They are two types of macro substitution.

1 : Macro substitution without arguments.

2 : Macro substitution with arguments.

1 : Macro substitution without arguments : It is a process to substitute the constant or value in the place of an identifier. It is possible to achieve this with the help of directive or macro definition statement **#define**.

Syntax : #define identifier constant or expression

Ex : #define PI 3.142

#define MAX_MARKS 100

#define MIN_MARKS 35

Ex :

```
#include <stdio.h>
 #define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'
 void main()
{
   printf("value of height    : %d \n", height );
   printf("value of number : %f \n", number );
   printf("value of letter : %c \n", letter );
   printf("value of letter_sequence : %s \n", letter_sequence);
   printf("value of backslash_char  : %c \n", backslash_char);
 }
```

OUTPUT:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
```

value of backslash_char : ?

Ex : **Example of Macro substitution**

```
#include<stdio.h>
#define PI 3.142
void main()
 {
    int r;
    float area;
    printf("Enter the radius of circle");
    scanf("%d",&r);
    area=PI*r*r;
    printf("the area of a circle is%d",area);
}
```

Example of Macro definition with expressions

```
  #define A (20*10)
#define  B (200-100)
void main()
{
   int div;
   div=A/B;
   printf("the division of two numbers%d",div);
}
```

Example of Macro definition with conditional expression

```
#define IFCONDITION if(a>b)
#define  PRINT printf("the value of a is the greatest no")
void main()
{
   int a=100,b=50;
   IFCONDITION
     PRINT;
}
```

Macro Substitution with Arguments :

Syntax : #define identifier(var1,var2,va3,….varn)string

Where **identifier** is the name of macro function with the list of macro formal parameters **var1,var2,var3,…varn** like the formal parameters in a function definition.

Ex :

```
#define  PROD(x)  (x*x)
void main()
{
    int a,mul;
    printf("enter the value of a");
    scanf("%d",&a);
    mul=PROD(a);
    printf("The multification of two numbers%d",mul);
}
```

2 : FILE INCLUSION : A copying of one file to another files into program.

Ex : File inclusion of an external file "add.c".

```
#include<stdio.h>
#include add.c
void main()
{
    void add();    //FUNCTION PROTOTYPE/ DECLARATION.
    add();         //FUNCTION CALLING
}
```

The file **add1.c** contains the function definition as follows.

```
void add()
{
    int a,b,c;
    printf("enter two numbers");
    scanf(%d%d",&a,&b);
   c=a+b;
   printf("c value is:%d",c);
}
```

3 : CONDITIONAL COMPILATION DIRECTIVES : C preprocessor also supports number of conditional compilation directives as

1 : #undef : Undefined a macro

2 : #ifdef : Tests for a macro definition.

3 : #endif : Specifies the end of #if.

4 : #if : Tests compile-time condition.

5 : #else : Specifies alternative when #if test fails.

These are used to select a particular segment of code for compilation depending on the condition.

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

- "#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.
- Otherwise, "else" clause statements are included in source file for compilation and execution.

Ex:

```
 #include <stdio.h>

#define RAJU 100

int main()

{

  #ifdef RAJU

  printf("RAJU is defined. So, this line will be added in " \

       "this C file\n");

  #else

  printf("RAJU is not defined\n");

  #endif

  return 0;

}
```

OUTPUT:

RAJU is defined. So, this line will be added in this C file

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

- #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.

Ex:

```
#include <stdio.h>
#define RAJU 100
int main()
{
  #ifndef SELVA
  {
    printf("SELVA is not defined. So, now we are going to " \
          "define here\n");
    #define SELVA 300
  }
  #else
  printf("SELVA is already defined in the program");

  #endif
  return 0;

}
```

OUTPUT:

SELVA is not defined. So, now we are going to define here

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

- "If" clause statement is included in source file if given condition is true.
- Otherwise, else clause statement is included in source file for compilation and execution.

Ex:

```
#include <stdio.h>
#define a 100
int main()
{
  #if (a==100)
  printf("This line will be added in this C file since " \
        "a \= 100\n");
  #else
  printf("This line will be added in this C file since " \
        "a is not equal to 100\n");
  #endif
  return 0;
}
```

OUTPUT:

This line will be added in this C file since a = 100

EXAMPLE PROGRAM FOR UNDEF IN C LANGUAGE:

This directive undefines existing macro in the program.

Ex:

```
#include <stdio.h>
 #define height 100
void main()
{
   printf("First defined value for height    : %d\n",height);
   #undef height          // undefining variable
   #define height 600     // redefining the same for new value
   printf("value of height after undef \& redefine:%d",height);
}
```

OUTPUT:

First defined value for height : 100

value of height after undef & redefine : 600

EXAMPLE PROGRAM FOR PRAGMA IN C LANGUAGE:

Pragma is used to call a function before and after main function in a C program.

Ex:

```
#include <stdio.h>
 void function1( );
void function2( );
#pragma startup function1
#pragma exit function2
```

```
int main( )
{
  printf ( "\n Now we are in main function" ) ;
  return 0;
}
 void function1( )
{
  printf("\nFunction1 is called before main function call");
}
 void function2( )
{
  printf ( "\nFunction2 is called just before end of " \
         "main function" ) ;"
}
```

OUTPUT:

Function1 is called before main function call

Now we are in main function

Function2 is called just before end of main function

Ex :

```
#define TEST 1
void main()
{
    #ifdef  TEST
    {
        printf(“This is compiled”);
```

```
        }
        #else
        {
            printf("This is not compiled");
        }
        #endif
    }
```

Ex2 : :

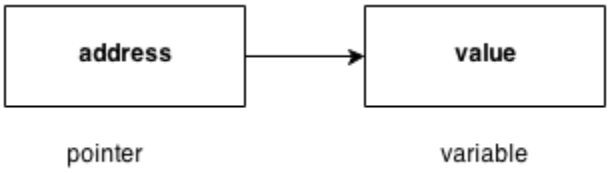
```
#define FLAG 1
char ch;
void main()
{
    #if  FLAG
    {
        ch='t';
        printf("This is compiled");
    }
    #else
    {
        ch='f';
        printf("This is not compiled");
    }
    #endif
}
```

POINTERS:

Introduction

Definition:

> Pointer is a variable that stores/hold address of another variable of same data type/ t is also known as locator or indicator that points to an address of a value. A pointer is a derived data type in C



Benefit 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 and the program execution time.
- It allows C to support dynamic memory management.

Declaration of Pointer

data_type* pointer_variable_name;

int* p;

Note: void type pointer works with all data types, but isn't used often.

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

int a = 10 ;

int *ptr ; *//pointer declaration*

ptr = &a ; *//pointer initialization*

or,

int *ptr = &a ; *//initialization and declaration together*

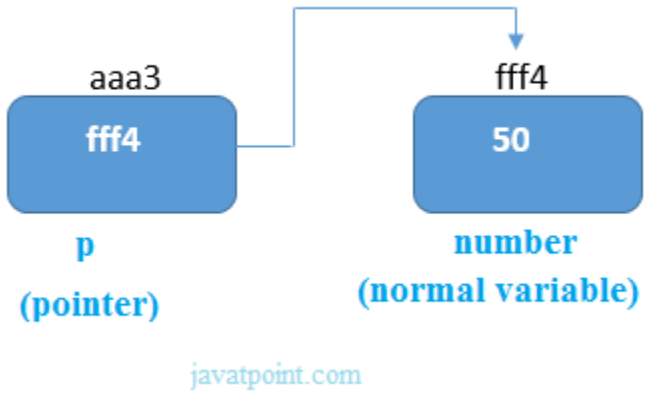
Note:Pointer variable always points to same type of data.

float a;

int *ptr;

ptr = &a; //ERROR, type mismatch

Above statement defines, p as pointer variable of type int. **Pointer example**



As you can see in the above figure, pointer variable stores the address of number variable i.e. fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), **we can print the value of pointer variable p.**

Reference operator (&) and Dereference operator (*)

& is called reference operator. It gives you the address of a variable. There is another operator that gets you the value from the address, it is called a dereference operator (*).

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	address of operator	determines the address of a variable.
* (asterisk sign)	indirection operator	accesses the value at the address.

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** *.

```
int a,*p;

a = 10;

p = &a;

printf("%d",*p);    //this will print the value of a.

printf("%d",*&a);  //this will also print the value of a.

printf("%u",&a);  //this will print the address of a.

printf("%u",p);  //this will also print the address of a.

printf("%u",&p);  //this will also print the address of p.
```

KEY POINTS TO REMEMBER ABOUT POINTERS IN C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

Example:

```
#include <stdio.h>
#include <conio.h>
void main(){
int number=50;
int *p;
clrscr();
p=&number;//stores the address of number variable
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of p variable is %d \n",*p);
getch();
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Example:

```
#include <stdio.h>
int main()
{
  int *ptr, q;
  q = 50;
```

```
   /* address of q is assigned to ptr */
   ptr = &q;
   /* display q's value using ptr variable */
   printf("%d", *ptr);
   return 0;
}
```

Output

50

Example:

```
#include <stdio.h>
int main()
{
   int var =10;
   int *p;
   p= &var;
   printf ( "\n Address of var is: %u", &var);
   printf ( "\n Address of var is: %u", p);
   printf ( "\n Address of pointer p is: %u", &p);
   /* Note I have used %u for p's value as it should be an address*/
   printf( "\n Value of pointer p is: %u", p);
   printf ( "\n Value of var is: %d", var);
   printf ( "\n Value of var is: %d", *p);
   printf ( "\n Value of var is: %d", *( &var));
}
```

Output:

Address of var is: 00XBBA77

Address of var is: 00XBBA77

Address of pointer p is: 77221111

Value of pointer p is: 00XBBA77

Value of var is: 10

Value of var is: 10

Value of var is: 10

NULL Pointer

A pointer that is not assigned any value but NULL is known as NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value.

Or

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.int *p=NULL;

Note: The NULL pointer is a constant with a value of zero defined in several standard libraries/ in most the libraries, the value of pointer is 0 (zero)

Example:

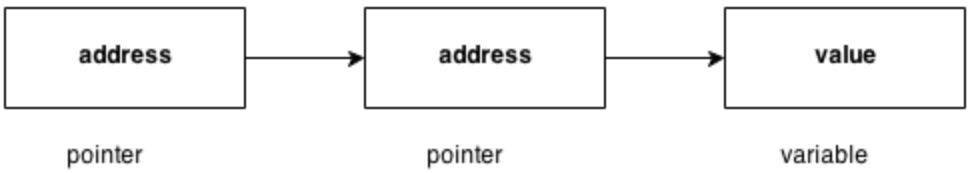
The value of ptr is 0

Pointers for Inter-Function Communication

Pointers to Pointers

Pointers can point to other pointers /pointer refers to the address of another pointer.

pointer can point to the address of another pointer which points to the address of a value.

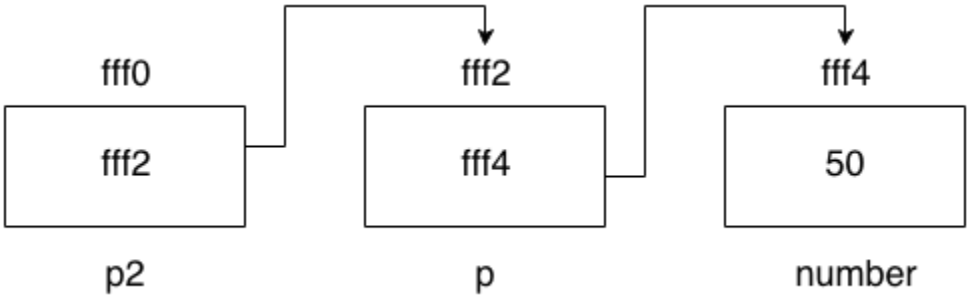


syntax of pointer to pointer

```
int **p2;
```

pointer to pointer example

Let's see an example where one pointer points to the address of another pointer.



Example:

```
#include <stdio.h>
#include <conio.h>
void main(){
int number=50;
int *p;//pointer to int
int **p2;//pointer to pointer
clrscr();
p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
```

```
printf("Value of *p variable is %d \n",*p);

printf("Address of p2 variable is %x \n",p2);

printf("Value of **p2 variable is %d \n",**p);

getch();

}
```

Output

Address of number variable is fff4

Address of p variable is fff4

Value of *p variable is 50

Address of p2 variable is fff2

Value of **p variable is 50

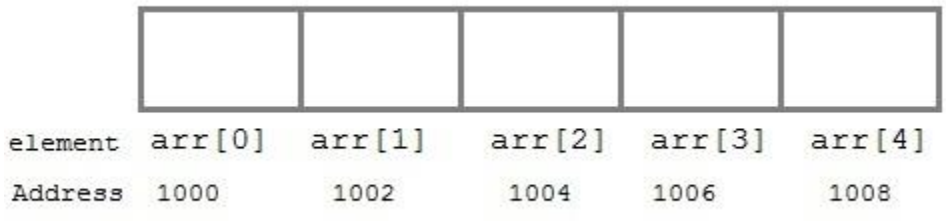
Arrays and Pointers

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address which gives location of the first element 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 byte, the five element 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.

arr *is equal to* **&arr[0]** // by default

We can declare a pointer of type int to point to the array arr.

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

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

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.**

Relation between Arrays and Pointers

Consider an array:

int arr[4];

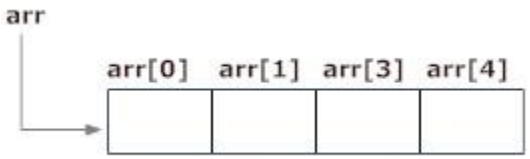


Figure: Array as Pointer

In C programming, name of the array always points to address of the first element of an array.

In the above example**, arr and &arr[0]** points to the address of the first element.

&arr[0] is equivalent to arr

Since, the addresses of both are the same, the values of arr and &arr[0] are also the same.

arr[0] is equivalent to *arr (value of an address of the pointer)

Similarly,

&arr[1] is equivalent to (arr + 1) AND, arr[1] is equivalent to *(arr + 1).

&arr[2] is equivalent to (arr + 2) AND, arr[2] is equivalent to *(arr + 2).

&arr[3] is equivalent to (arr + 3) AND, arr[3] is equivalent to *(arr + 3).

.

.

&arr[i] is equivalent to (arr + i) AND, arr[i] is equivalent to *(arr + i).

Example: Program to find the sum of six numbers with arrays and pointers

```
#include <stdio.h>
int main()
{
  int i, classes[6],sum = 0;
  printf("Enter 6 numbers:\n");
  for(i = 0; i < 6; ++i)
  {
     // (classes + i) is equivalent to &classes[i]
     scanf("%d",(classes + i));
     // *(classes + i) is equivalent to classes[i]
     sum += *(classes + i);
  }
  printf("Sum = %d", sum);
  return 0;
}
```

Output

Enter 6 numbers:

2

3

4

5

3

4

Sum = 21

Pointer Arithmetic and Arrays

pointer holds address of a value, so there can be arithmetic operations on the pointer variable.

There are four arithmetic operators that can be used on pointers:

- Increment(++)
- Decrement(--)
- Addition(+)
- Subtraction(-)

Increment pointer:

1. Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.
2. Incrementing Pointer Variable Depends Upon data type of the Pointer variable.

The formula of incrementing pointer is given below:

new_address= current_address + i * size_of(data type)

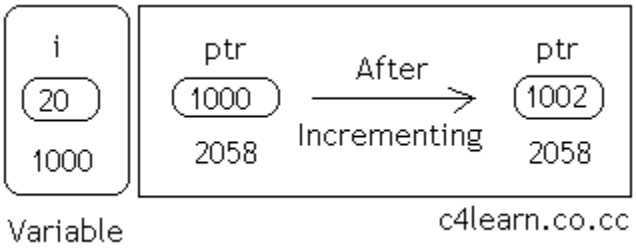
Three rules should be used to increment pointer

Address + 1 = Address

Address++ = Address

++Address = Address

Pictorial Representation :



Data Type	**Older Address stored in pointer**	**Next Address stored in pointer after incrementing (ptr++)**
int	1000	1002
float	1000	1004
char	1000	1001

Note :

32 bit

For 32 bit int variable, it will increment to 2 byte.

64 bit

For 64 bit int variable, it will increment to 4 byte.

Example:

```
#include <stdio.h>

void main(){

int number=50;
```

```
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p);
}
```

Output

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Decrement(--)

Like increment, we can decrement a pointer variable.

formula of decrementing pointer

```
new_address= current_address - i * size_of(data type)
```

Example:

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
```

```
printf("After decrement: Address of p variable is %u \n",p);
}
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

Addition(+)

We can add a value to the pointer variable.

formula of adding value to pointer

new_address= current_address + (number * size_of(data type))

Note:

32 bit

For 32 bit int variable, it will add 2 * number.

64 bit

For 64 bit int variable, it will add 4 * number.

Example:

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
 printf("Address of p variable is %u \n",p);
```

```
p=p+3;   //adding 3 to pointer variable

printf("After adding 3: Address of p variable is %u \n",p);

}
```

Output

```
Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312
```

Subtraction (-)

Like pointer addition, we can subtract a value from the pointer variable. The formula of subtracting value from pointer variable.

```
new_address= current_address - (number * size_of(data type))
```

Example:

```
#include <stdio.h>

void main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

 printf("Address of p variable is %u \n",p);

p=p-3; //subtracting 3 from pointer variable

printf("After subtracting 3: Address of p variable is %u \n",p);

}
```

Output

Address of p variable is 3214864300

After subtracting 3: Address of p variable is 3214864288

Passing an Array to a Function

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

1) **Formal parameters as a pointer –**

```
void myFunction(int *param) {
                .
                .
                .
            }
```

2) **Formal parameters as a sized array –**

```
void myFunction(int param[10]) {
                .
                .
                .
            }
```

3) **Formal parameters as an unsized array −**

```
void myFunction(int param[10]) {
                .
                .
                .
            }
```

Example1: pass an entire array to a function argument

```
#include <stdio.h>
/* function declaration */
double getAverage(int arr[], int size);
int main () {
  /* an int array with 5 elements */
  int balance[5] = {1000, 2, 3, 17, 50};
```

```
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;
    /* output the returned value */
   printf( "Average value is: %f ", avg );
       return 0;
}
double getAverage(int arr[], int size) {

   int i;
   double avg;
   double sum = 0;

   for (i = 0; i < size; ++i) {
     sum += arr[i];
   }

   avg = sum / size;

   return avg;
}
```

Output

Average value is: 214.400000

Example2: pass an entire array to a function argument

```
#include <stdio.h>
myfuncn( int *var1, int var2)
{
    for(int x=0; x<var2; x++)
    {
        printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
    }
}
```

```
int main()
{
    int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
    myfuncn(&var_arr, 7);
    return 0;
}
```

Output

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
```

Example: Call by value method –

```
#include <stdio.h>
disp( char ch)
{
  printf("%c ", ch);
}
int main()
{
  char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'I', 'j'};
  for (int x=0; x<=10; x++)
  {
     /* I'm passing each element one by one using subscript*/
     disp (arr[x]);
  }

  return 0;
}
```

Output:

```
a b c d e f g h i j
```

In this method of calling a function, the actual arguments gets copied into formal arguments. In this example actual argument(or parameter) is arr[x] and formal parameter is ch.

Example: Call by reference method: Using pointers

```
#include <stdio.h>

disp( int *num)

{

   printf("%d ", *num);

}

int main()

{

    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    for (int i=0; i<=10; i++)

    {

       /* I'm passing element's address*/

       disp (&arr[i]);

    }

    return 0;

}
```

Output:

1 2 3 4 5 6 7 8 9 0

Array of Pointers

An array of pointers would be an array that holds memory locations. An array of pointers is an indexed set of variables in which the variables are pointers (a reference to a location in memory).

Syntax:

data_type_name * variable name

Example

int *ptr[MAX];

Array alpha[]	**Pointer *a***
alpha[0]	*a
alpha[1]	*(a+1)
alpha[2]	*(a+2)
alpha[3]	*(a+3)
alpha[*n*]	*(a+*n*)

Example1:

```
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};

   int i, *ptr[MAX];

   for ( i = 0; i < MAX; i++) {
```

```
    ptr[i] = &var[i]; /* assign the address of integer. */
  }
    for ( i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }
  return 0;
}
```

Output

Value of var[0] = 10

Value of var[1] = 100

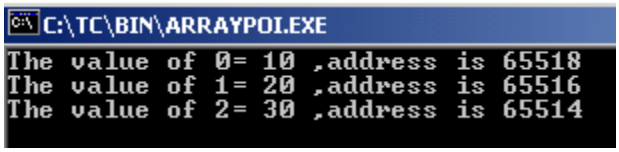
Value of var[2] = 200

Example2:

```
#include <stdio.h>
#include <conio.h>
main() {
  clrscr();
  int *array[3];
  int x = 10, y = 20, z = 30;
  int i;
  array[0] = &x;
  array[1] = &y;
  array[2] = &z;
  for (i=0; i< 3; i++) {
  printf("The value of %d= %d ,address is %u\t \n", i, *(array[i]),
  array[i]);
  }
  getch();
```

```
  return 0;
}
```

Output



Example3:

```
#include <stdio.h>

const int MAX = 4;

int main () {

   char *names[] = {

      "Zara Ali",

      "Hina Ali",

      "Nuha Ali",

      "Sara Ali"

   };

      int i = 0;

   for ( i = 0; i < MAX; i++) {

      printf("Value of names[%d] = %s\n", i, names[i] );

   }

   return 0;

}
```

Output:

Value of names[0] = Zara Ali

Value of names[1] = Hina Ali

Value of names[2] = Nuha Ali

Value of names[3] = Sara Ali

Example4:

```
#include <stdio.h>

int main()

{

 char *fruit[] = {

 "watermelon",

 "banana",

 "pear",

 "apple",

 "coconut",

 "grape",

 "blueberry"

 };

 int x;

 for(x=0;x<7;x++)

 puts(fruit[x]);
```

return(0);

}

Pointers to Void and to Functions

Pointers to Void

Note:

1. Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
2. Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer,character pointer.

A pointer variable declared using a particular data type can not hold the location address of variables of other data types. It is invalid and will result in a compilation error.

Ex:- char *ptr;

int var1;

ptr=&var1; // This is invalid because 'ptr' is a character pointer variable.

Here comes the importance of a "void pointer". A void pointer is nothing but a pointer variable declared using the reserved word in C 'void'.

Void Pointer Basics :

3. In C **General Purpose Pointer** is called as void Pointer.
4. It does not have any data type associated with it
5. It can store address of any type of variable
6. A void pointer is a C convention for a raw address.
7. The compiler has no idea what type of object a void Pointer really points to ?

Void pointer: A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type. **Special type of pointer** called void pointer or **general purpose pointer**.

Declaration of void pointer

void * pointer_name;

Void pointer example

```
void *ptr;    // ptr is declared as Void pointer

char cnum;

int inum;

float fnum;

ptr = &cnum;  // ptr has address of character data

ptr = &inum;  // ptr has address of integer data

ptr = &fnum;  // ptr has address of float data
```

Advantages of void pointers:

1) malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)

{

   // Note that malloc() returns void * which can be

   // typecasted to any type like int *, char *, ..
```

```
    int *x = malloc(sizeof(int) * n);
}
```

2) void pointers in C are used to implement generic functions in C.

Note:

1) void pointers cannot be dereferenced. For example the following program doesn't compile.

```
#include<stdio.h>
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

Output:

Compiler Error: 'void*' is not a pointer-to-object type

The following program compiles and runs fine.

```
#include<stdio.h>
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *(int *)ptr);
    return 0;
}
```

Output:

10

Summary : Void Pointer

Scenario	Behavior
When We assign address of integer variable to void pointer	Void Pointer Becomes Integer Pointer
When We assign address of character variable to void pointer	Void Pointer Becomes Character Pointer
When We assign address of floating variable to void pointer	Void Pointer Becomes Floating Pointer

Pointers to functions/ Function Pointers

- A pointer to a function points to the address of the executable code of the function.
- We can use pointers to call functions and to pass functions as arguments to other functions.
- We cannot perform pointer arithmetic on pointers to functions.
- The type of a pointer to a function is based on both the return type and parameter types of the function.
- A declaration of a pointer to a function must have the pointer name in parentheses.
- The function call operator () has a higher precedence than the dereference operator *. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type.

declare Pointer to function?

<function return type>(*<Pointer_name>)(function argument list)

For example –

For example:

1) int *f(int a); /* function f returning an int * */

In this declaration, f is interpreted as a function that takes an int as argument, and returns a pointer to an int.

2) double (*p2f)(double, char)

Here double is a return type of function, p2f is pointer name & (double, char) is an argument list for the function. Which means the first argument for this function should be double and the second one would be of char type.

Example:

```
#include<stdio.h>
int sum (int num1, int num2)
{
   return sum1+sum2;
}
int main()
{
   int (*f2p) (int, int);
   f2p = sum;
   int op1 = f2p (10, 13);
   int op2 = sum (10, 13);
   printf("Output 1 – for function call via Pointer: %d",op1);
   printf("Output2 – for direct function call: %d", op2);
   return 0;
}
```

Output:

Output 1 – for function call via Pointer: 23

Output2 – for direct function call: 23

You would have noticed that the output of both the statements is same – f2p(10, 13) == sum(10, 13)

which means in generic sense you can write it out as:

pointer_name(argument list) == function(same argument list)

wherein pointer_name is declared as:

return_type(*pointer_name)(argument list);

pointer_name = function_name(argument list);

Memory Allocation Functions

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*.

Or

The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h**.

Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

Note: Dynamic memory allocation related function can be applied for any data type that's why dynamic memory allocation related functions return void*.

Memory Allocation Process

Global variables, **static** variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in area called **Stack**. The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.

malloc()

malloc stands for "memory allocation".

The malloc() function allocates single block of requested memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. It doesn't initialize memory at execution time, so it has garbage value initially. If it fails to locate enough space (memory) it returns a NULL pointer.

syntax

ptr=(cast-type*)malloc(byte-size)

Example

```
int *x;
x = (int*)malloc(100 * sizeof(int));    //memory space allocated to variable x
free(x);                 //releases the memory allocated to variable x
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

calloc()

calloc stands for "contiguous allocation".

Calloc() is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**. The calloc() function allocates multiple block of requested memory.

It initially initialize (sets) all bytes to zero.If it fails to locate enough space(memory) it returns a NULL pointer. The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size.

Syntax

ptr = (cast-type*)calloc(n/number, element-size);

calloc() required 2 arguments of type count, size-type.

Count will provide number of elements; size-type is data type size

Example

```
int*arr;
arr=(int*)calloc(10, sizeof(int));        // 20 byte
cahr*str;
str=(char*)calloc(50, siceof(char));  // 50 byte
```

Example

```
struct employee
{
 char *name;
 int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) calloc(num, sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

Diffrence between malloc() and calloc()

calloc()	**malloc()**
calloc() initializes the allocated memory with 0 value.	malloc() initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax : (cast_type *)calloc(blocks , size_of_block);	**Syntax :** (cast_type *)malloc(Size_in_bytes);

realloc(): changes memory size that is already allocated to a variable.

Or

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size. By using realloc() we can create the memory dynamically at middle stage. Generally by using realloc() we can reallocation the memory. Realloc() required 2 arguments of type void*, size_type. Void* will indicates previous block base address, size-type is data type size. Realloc() will creates the memory in bytes format and initial value is garbage.

syntax

ptr=realloc(ptr, new-size)

Example

```
int *x;
x=(int*)malloc(50 * sizeof(int));
x=(int*)realloc(x,100); //allocated a new memory to variable x
```

Example

```
void*realloc(void*, size-type);
int *arr;
arr=(int*)calloc(5, sizeof(int));
.....
........
....
arr=(int*)realloc(arr,sizeof(int)*10);
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
       printf("%u\t",ptr + i);
   printf("\nEnter new size of array: ");
   scanf("%d", &n2);
   ptr = realloc(ptr, n2);
   for(i = 0; i < n2; ++i)
       printf("%u\t", ptr + i);
   return 0;
}
```

free()

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Or

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

Syntax:

free(ptr);

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &num);

   ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)
   {
      scanf("%d", ptr + i);
```

```
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Command-Line Arguments:

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function

Syntax:

int main(**int** argc, **char** *argv[])

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

Example1

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
```

```
}
```

Output

Example2

```
#include <stdio.h>
void main(int argc, char *argv[] )  {
  printf("Program name is: %s\n", argv[0]);
     if(argc < 2){
     printf("No argument passed through command line.\n");
   }
   else{
     printf("First argument is: %s\n", argv[1]);
   }
}
```

Output

program.exe hello

Program name is: program

First argument is: hello

Note

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

Example

./program "hello c how r u"

Program name is: program

First argument is: hello c how r u

You can write your program to print all the arguments. In this program, we are printing only argv[1], that is why it is printing only one argument.

Example3

```
#include<stdio.h>
#include<conio.h>
void main(int argc, char* argv[])
{
int i;
clrscr();
printf("Total number of arguments: %d",argc);
for(i=0;i< argc;i++)
{
printf("\n %d argument: %s",i,argv[i]);
getch();
}
}
```

Output

C:/TC/BIN>TCC mycmd.c

C:/TC/BIN>mycmd 10 20

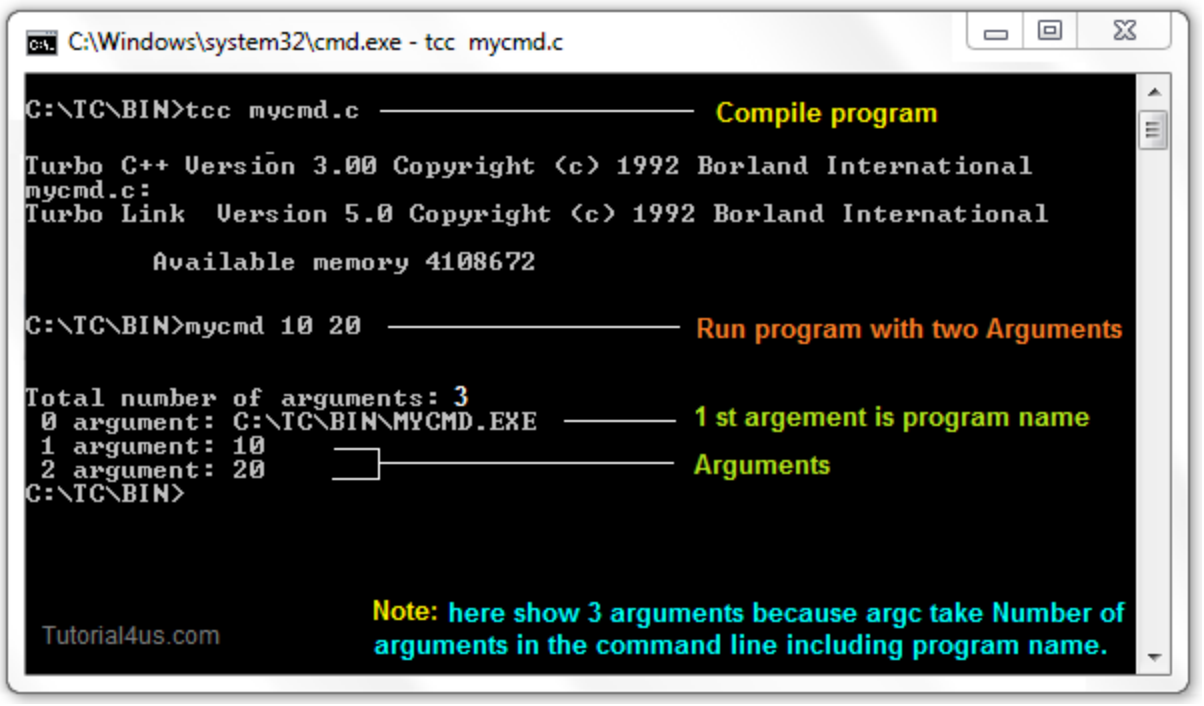
Number of Arguments: 3

0 arguments c:/tc/bin/mycmd.exe

1 arguments: 10

2 arguments: 20

Note: In above output we passed two arguments but is show "Number of Arguments: 3" because **argc** take Number of arguments in the command line including program name. So here two arguments and one program name (mycmd.exe) total 3 arguments.



Example4:

```
#include<stdio.h>
#include<conio.h>
void main(int argc, char* argv[])
{
clrscr();
  printf("\n Program name  : %s \n", argv[0]);
  printf("1st arg  : %s \n", argv[1]);
  printf("2nd arg  : %s \n", argv[2]);
  printf("3rd arg  : %s \n", argv[3]);
  printf("4th arg  : %s \n", argv[4]);
  printf("5th arg  : %s \n", argv[5]);
getch();
}
```

Output

C:/TC/BIN>TCC mycmd.c

C:/TC/BIN>mycmd this is a program

Program name : c:/tc/bin/mycmd.c

1st arg : this

2nd arg : is

3rd arg : a

4th arg : program

5th arg : (null)

Explanation: In the above example.

argc = 5

argv[0] = "mycmd"

argv[1] = "this"

argv[2] = "is"

argv[3] = "a"

argv[4] = "program"

argv[5] = NULL

Why command line arguments program not directly run form TC IDE

Command line arguments related programs are not execute directly from TC IDE because arguments can not be passed.

Edit Command Line Argument Program

To Edit the Command Line Argument Program use **edit** Command.

Syntax

C:/cprogram>edit mycmd.c

UNIT IV

STRUCTURES, UNIONS, ENUMERATIONS AND TYPEDEF

Structure Definition

Structure is a user defined data type which hold or store heterogeneous/different types data item or element in a single variable. It is a Combination of primitive and derived data type.

or

A structure is a collection of one or more data items of different data types, grouped together under a single name.

Variables inside the structure are called members of structure.

Each element of a structure is called a member.

struct keyword is used to define/create a structure. **struct** define a new data type which is a collection of different type of data.

Syntax

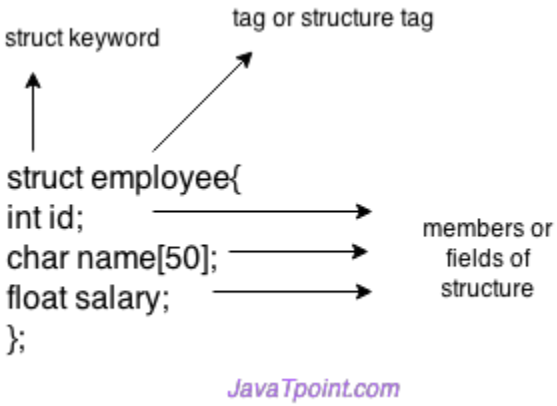
```
struct structure_name /tag name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type member n;
};
```

Note: Don't forget the semicolon }; in the ending line.

Example

```
struct employee
{   int id;
    char name[50];
    float salary;
};
```

Here, **struct** is the keyword, **employee** is the tag name of structure; **id**, **name** and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



Syntax to create structure variable

struct tagname/structure_name variable;

Declaring structure variable

We can declare variable for the structure, so that we can access the member of structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function/ Declaring Structure variables separately
2. By declaring variable at the time of defining structure/ Declaring Structure Variables with Structure definition

1st way:

Let's see the example to declare structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{   int id;
```

```
    char name[50];
    float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

2nd way:

Let's see another way to declare variable at the time of defining structure.

```
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

Which approach is good

But if no. of variable are not fixed, use 1st approach. It provides you flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare variable in main() function.

Structure Initialization

structure variable can also be initialized at compile time.

```
struct Patient
{
 float height;
 int weight;
```

```
 int age;
};
```

struct Patient **p1** = { 180.75 , 73, 23 }; *//initialization*

or

struct patient p1;

p1.height = 180.75; ***//initialization of each member separately***

p1.weight = 73;

p1.age = 23;

Accessing Structures/ Accessing members of structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

When the variable is normal type then go for struct to member operator.

When the variable is pointer type then go for pointer to member operator.

Any member of a structure can be accessed as:

structure_variable_name.member_name

Example

```
struct book
{
char name[20];
char author[20];
int pages;
};
```

struct book b1;

for accessing the structure members from the above example

b1.name, b1.author, b1.pages:

Example

```
struct emp
{
int id;
char name[36];
int sal;
};
sizeof(struct emp)       // --> 40 byte (2byte+36byte+2byte)
```

Example of Structure in C

```
#include<stdio.h>
#include<conio.h>
struct emp
{
int id;
char name[36];
float sal;
};
void main()
{
struct emp e;
clrscr();
printf("Enter employee Id, Name, Salary: ");
```

```
scanf("%d",&e.id);
scanf("%s",&e.name);
scanf("%f",&e.sal);

printf("Id: %d",e.id);
printf("\nName: %s",e.name);
printf("\nSalary: %f",e.sal);
getch();
}
```

Output

Output: Enter employee Id, Name, Salary: 5 Spidy 45000

Id : 05

Name: Spidy

Salary: 45000.00

Example

```
#include <stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
```

```
    e1.id=101;
    strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
    //printing first employee information
    printf( "employee 1 id : %d\n", e1.id);
    printf( "employee 1 name : %s\n", e1.name);
    return 0;
}
```

Output:

employee 1 id : 101

employee 1 name : Sonoo Jaiswal

Difference Between Array and Structure

1	Array is collection of homogeneous data.	Structure is the collection of heterogeneous data.
2	Array data are access using index.	Structure elements are access using . operator.
3	Array allocates static memory.	Structures allocate dynamic memory.
4	Array element access takes less time than structures.	Structure elements takes more time than Array.

Nested Structures

structure can have another structure as a member. There are two ways to define nested structure in c language:

1. By separate structure
2. By Embedded structure

1) Separate structure

We can create 2 structures, but dependent structure should be used inside the main structure as a member. Let's see the code of nested structure.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};

struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

2) Embedded structure

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
     int dd;
     int mm;
     int yyyy;
   }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of nested structure by Outer_Structure.Nested_Structure.member as given below:

e1.doj.dd

e1.doj.mm

e1.doj.yyyy

Arrays of Structures

Array of structures to store much information of different data types. Each element of the array representing a **structure** variable. The array of structures is also known as collection of structures.

Ex : if you want to handle more records within one structure, we need not specify the number of structure variable. Simply we can use array of structure variable to store them in one structure variable.

Example : struct employee emp[5];

Example of structure with array that stores information of 5 students and prints it.

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

struct student{

int rollno;

char name[10];

};

void main(){

int i;

struct student st[5];

clrscr();
```

```
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
  printf("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
getch();
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz
```

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz

Structures and Functions

A structure can be passed as a function argument just like any other variable. This raises a few practical issues.

PASSING STRUCTURE TO FUNCTION IN C:

It can be done in below 3 ways.

1. Passing structure to a function by value
2. Passing structure to a function by address(reference)
3. No need to pass a structure – Declare structure variable as global

.

The general format of sending a copy of a structure to the called function is:

```
Function_name(structure_variable_name);
```

The called function takes the following form:

```
data_type function_name(struct tag_name var)
{
```

```
        ----------
        ----------
        return(exp);
    }
```

PASSING STRUCTURE TO FUNCTION IN C BY VALUE:

```
#include <stdio.h>
#include <string.h>
struct student
{
    int id;
    char name[20];
    float percentage;
};
 void func(struct student record);
 int main()
{
    struct student record;
     record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
     func(record);
    return 0;
}
 void func(struct student record)
```

```
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

Output

Id is: 1

Name is: Raju

Percentage is: 86.500000

PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:

```
#include <stdio.h>
#include <string.h>
 struct student
{
        int id;
        char name[20];
        float percentage;
};
 void func(struct student *record);
 int main()
{
       struct student record;
        record.id=1;
       strcpy(record.name, "Raju");
       record.percentage = 86.5;
```

```
    func(&record);
    return 0;
}
 void func(struct student *record)
{
    printf(" Id is: %d \n", record->id);
    printf(" Name is: %s \n", record->name);
    printf(" Percentage is: %f \n", record->percentage);
}
```

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:

```
#include <stdio.h>
#include <string.h>
 struct student
{
     int id;
     char name[20];
     float percentage;
};
struct student record; // Global declaration of structure
 void structure_demo();
 int main()
{
     record.id=1;
```

```
        strcpy(record.name, "Raju");
        record.percentage = 86.5;

        structure_demo();
        return 0;
}

void structure_demo()
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

Passing a copy of entire structure to a function

```
struct std
{
int no;
float avg;
};
struct std a;
void fun(struct std p);
void main()
```

```
{
clrscr();
a.no=12;
a.avg=13.76;
fun(a);
getch();
}
void fun(struct std p)
{
printf("number is%d\n",p.no);
printf("average is%f\n",p.avg);
}
```

Passing Structures through Pointers

Example

```
#include <stdio.h>
#include <string.h>
 struct student
{
    int id;
    char name[30];
    float percentage;
```

```
};
 int main()
{
    int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student *ptr;
     ptr = &record1;
        printf("Records of STUDENT1: \n");
        printf("  Id is: %d \n", ptr->id);
        printf("  Name is: %s \n", ptr->name);
        printf("  Percentage is: %f \n\n", ptr->percentage);

    return 0;
}
```

OUTPUT:

Records of STUDENT1:
Id is: 1
Name is: Raju

Percentage is: 90.500000

Self-referential Structures

A structure consists of at least a pointer member pointing to the same structure is known as a self-referential structure. A self referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

Syntax : struct tag_name

```
{
    type member1;
    type membere2;
     :        :
     :        :
    typeN memberN;
    struct tag_name *name;
 }
```

Where *name refers to the name of a pointer variable.

Ex:

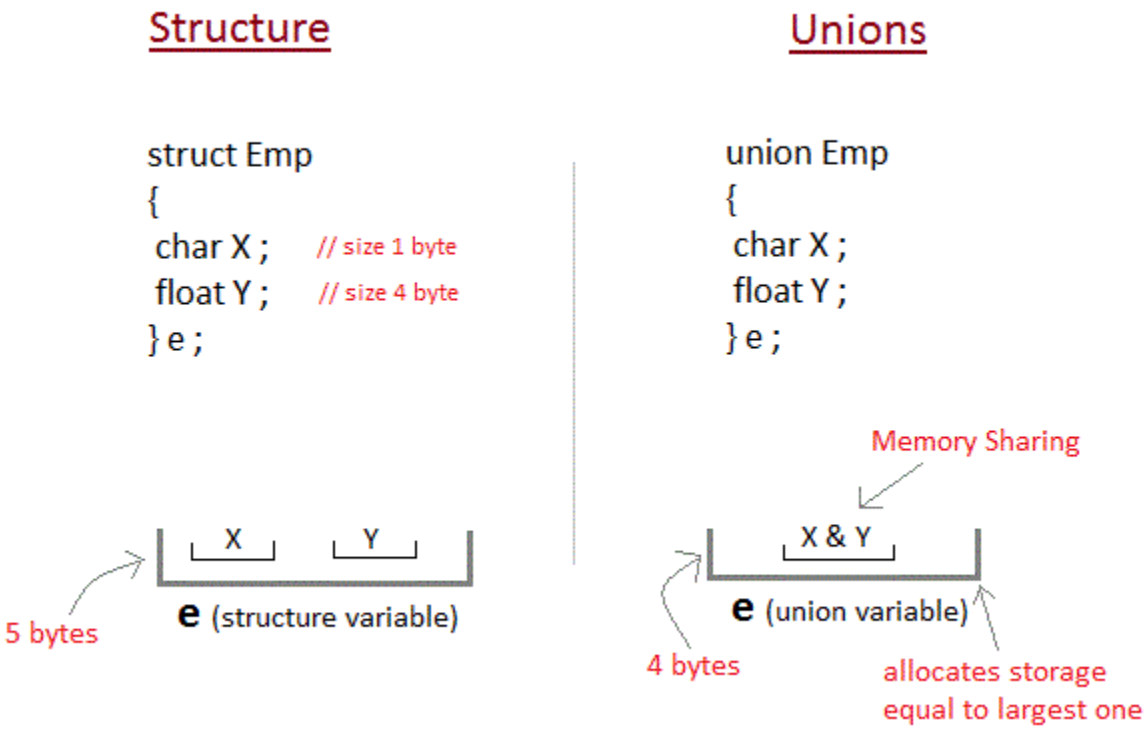
```
struct emp
{
int code;
struct emp *name;
}
```

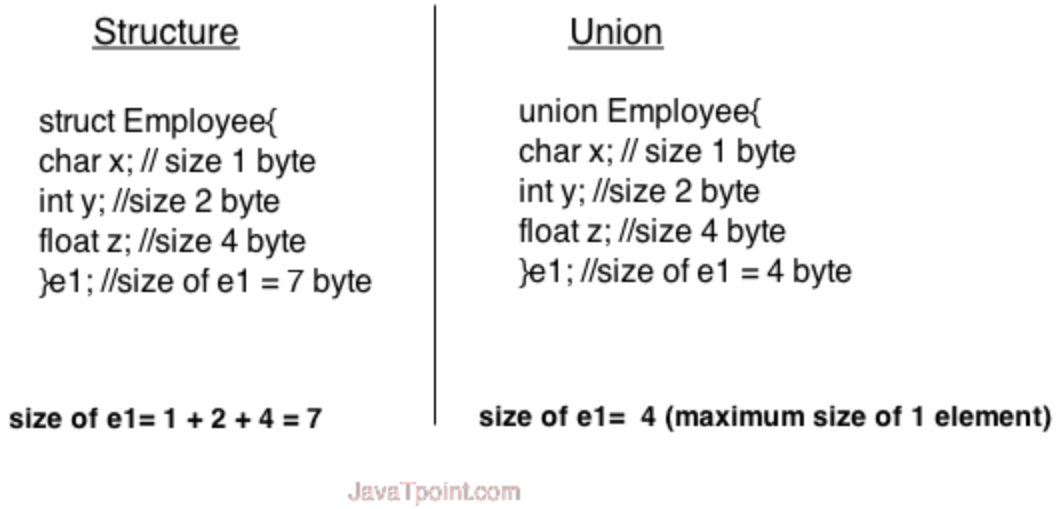
Unions

A union is a special data type available in C that allows to store different data types in the same memory location.

Unions are conceptually similar to **structures**. The syntax of **union** is also similar to that of structure. The only difference is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** use a single shared memory location which is equal to the size of its largest data member.

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. Where **as** Structure allocates storage space for all its members separately.





syntax

```
union union_name
{
   data_type member1;
   data_type member2;
  .
  .
   data_type memeberN;
};
```

Example

```
union employee
{   int id;
   char name[50];
   float salary;
};
```

Example

```
#include <stdio.h>
#include <string.h>
union employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for union
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
   return 0;
}
```

Output:

employee 1 id : 1869508435

employee 1 name : Sonoo Jaiswal

As you can see, id gets garbage value because name has large memory size. So only name will have actual value.

Example

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

```
{
 int a;
 float b;
 char ch;
};
int main( )
{
 union item it;
 it.a = 12;
 it.b = 20.2;
 it.ch='z';
 clrscr();
 printf("%d\n",it.a);
 printf("%f\n",it.b);
 printf("%c\n",it.ch);
 getch();
 return 0;
}
```

Output

```
-26426
20.1999
z
```

As you can see here, the values of **a** and **b** get corrupted and only variable **c** prints the expected result. Because in **union**, the only member whose value is currently stored will have the memory.

Difference between Structure and Union

	Structure	Union
1	For defining structure use struct keyword.	For defining union we use union keyword
2	Structure occupies more memory space than union.	Union occupies less memory space than Structure.
3	In Structure we can access all members of structure at a time.	In union we can access only one member of union at a time.
4	Structure allocates separate storage space for its every members.	Union allocates one common storage space for its all members. Union find which member need more memory than other member, then it allocate that much space

Bit-Fields

Syntax

```
struct {
   type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field −

Elements	Description
type	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
member_name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

```
struct {
   unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example –

```
#include <stdio.h>
#include <string.h>
struct {
   unsigned int age : 3;
} Age;
int main( ) {
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );
   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );
   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );
   return 0;
}
```

Output

```
Sizeof( Age ) : 4
```

Age.age : 4

Age.age : 7

Age.age : 0

typedef

The typedef is a keyword that allows the programmer to create a new data type name for an existing data type. So, the purpose of typedef is to redefine the name of an existing variable type.

Syntax

typedef datatype alias_name;

Example of typedef

```
#include<stdio.h>

#include<conio.h>

typedef int Intdata; // Intdata is alias name of int

void main()

{

int a=10;

Integerdata b=20;

typedef Intdata Integerdata; // Integerdata is again alias name of Intdata

Integerdata s;

clrscr();

s=a+b;

printf("\n Sum:= %d",s);

getch();

}
```

Output

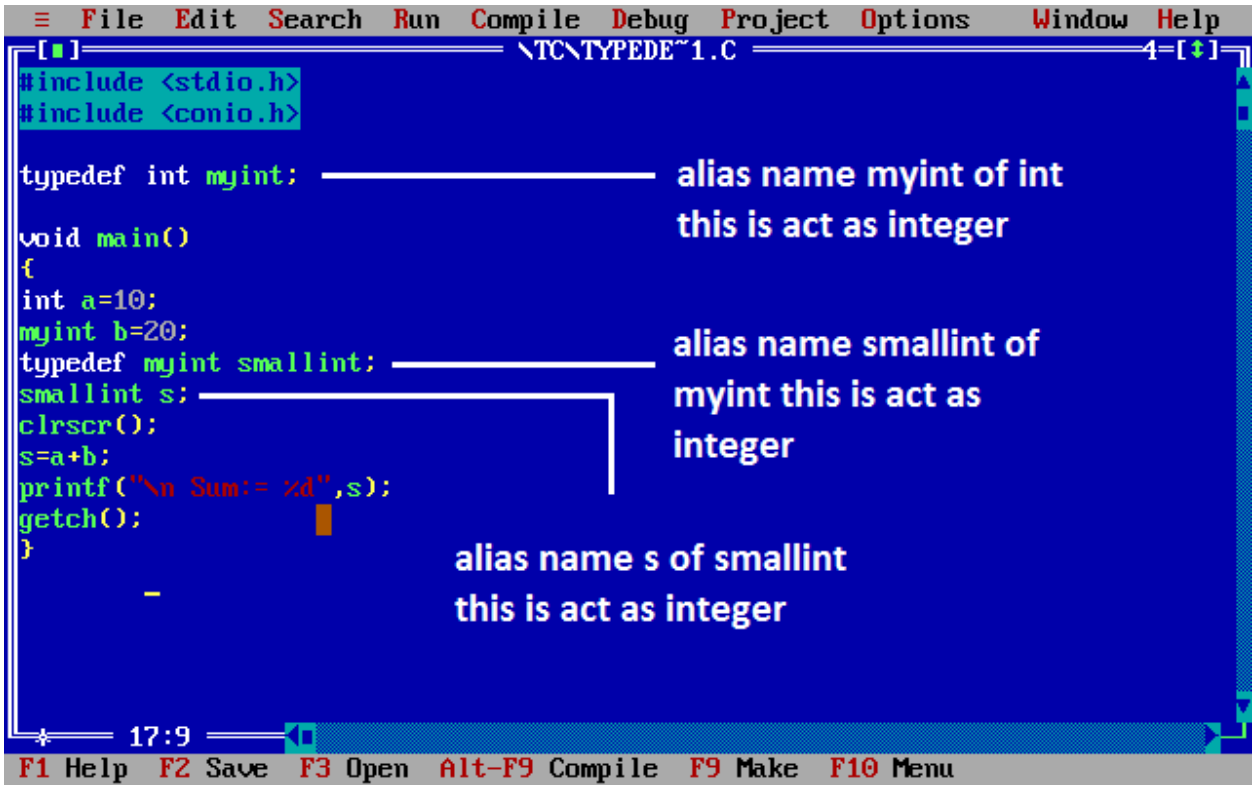
Sum: 20

Advantages of typedef :

1 : Provides a meaningful way of declaring the variable.

2 : Increase the readability of the program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  typedef int digits;
  digits a,b,sum;
  clrscr();
  printf("Enter a and b values:");
  scanf("%d%d",&a,&b);
  sum=a+b;
  printf("The sum is:%d",sum);
  getch();
}
```



Note: By using typedef only we can create the alias name and it is under control of compiler.

Application of typedef

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

typedef struct

{

type member1;

type member2;

type member3;

} **type_name** ;

Here **type_name** represents the stucture definition associated with it. Now this **type_name** can be used to declare a variable of this stucture type.

type_name t1, t2 ;

Example of structure definition using typedef

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
 typedef struct employee
{
 char  name[50];
 int   salary;
} emp ;
 void main( )
{
 emp e1;
 printf("\nEnter Employee record\n");
 printf("\nEmployee name\t");
 scanf("%s",e1.name);
 printf("\nEnter Employee salary \t");
 scanf("%d",&e1.salary);
 printf("\nstudent name is %s",e1.name);
 printf("\nroll is %d",e1.salary);
 getch();
}
```

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers * binds to the right and not the left.

int* x, y ;

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain integer.

typedef int* **IntPtr** ;

IntPtr x, y, z;

But if we use **typedef** like in above example, we can declare any number of pointers in a single statement.

NOTE : If you do not have any prior knowledge of pointers, do study Pointers first.

Enumerations

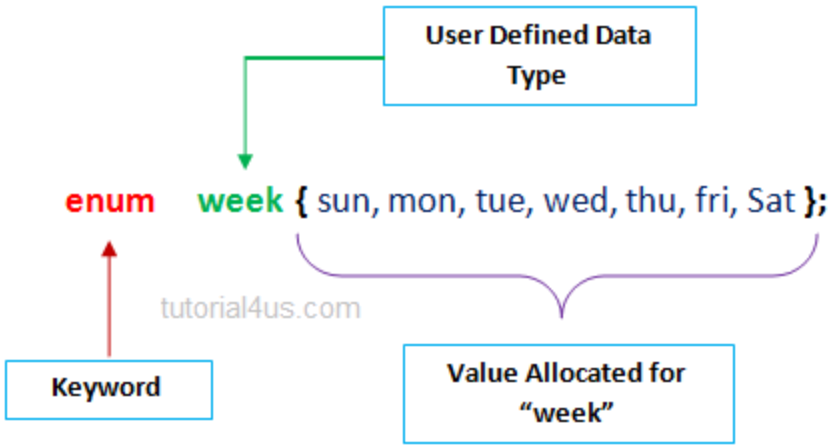
An **enum** is a keyword, it is an user defined data type. All properties of integer are applied on Enumeration data type so size of the enumerator data type is 2 byte. It work like the Integer.

It is used for creating an user defined data type of integer. Using enum we can create sequence of integer constant value.

Syntax

enum tagname {value1, value2, value3,....};

- In above syntax **enum** is a keyword. It is a user defiend data type.
- In above syntax tagname is our own variable. tagname is any variable name.
- value1, value2, value3,.... are create set of enum values.



It is start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list. If constant one value is not initialized then by default sequence will be start from zero and next to generated value should be previous constant value one.

Example of Enumeration in C

```
#include<stdio.h>

#include<conio.h>

enum ABC {x,y,z};

void main()

{

int a;

clrscr();

a=x+y+z;        //0+1+2

printf("Sum: %d",a);

getch();

}
```

Output

Sum: 3

Example of Enumeration in C

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

```
#include<conio.h>
enum week {sun, mon, tue, wed, thu, fri, sat};
void main()
{
 enum week today;
 today=tue;
 printf("%d day",today+1);
 getch();
}
```

Output

3 day

Example of Enumeration in C

```
#include<stdio.h>
#include<conio.h>
enum week {sun, mon, tue, wed, thu, fri, sat};
void main()
{
for(i=sun; i<=sat; i++)
{
 printf("%d ",i);
}
getch();
}
```

Output

In above code replace sun, mon, tue,.... with Equivalent numeric value 0, 1, 2,...

UNIT – V

FILE

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

File I/O:-

Sometimes it is necessary to store the data in a manner that can be later retrieved and displayed either in a part or in whole. This medium is usually a "file" on the disk. File I/O can be handled by using different functions.

a) Formatted functions:- The file input function fscanf( ) and the file output function fprintf( ) are called formatted file I/O functions.

b)Unformatted functions:- The input functions like getc( ), getw( ), and fread( ) are called unformatted file input functions and putc( ), putw( ), and fwrite( ) functions are unformatted file output functions. Each and every function is having its own syntax and meaning.

File streams:- Stream is either reading or writing of data. The streams are designed to allow the user to access the files efficiently. A stream is a file or physical device like key board, printer, monitor, etc., The FILE object uses these devices. When a C program is started, the operating system is responsible for opening three streams: standard input stream (**stdin**), standard output stream (**stdout**), standard error(**stderr**).Normally the stdin is connected to the keyboard, the stdout and stderr are connected to the monitor.

Files

File is a collection of bytes that is stored on secondary storage devices like Hard disk.

OR

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 ready made structure.

Note:

All files related function are available in **stdio.h** header file.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations

In C, you can perform four major operations on the file, either text or binary:

- Naming a file/Creation of new file
- Opening an existing file
- Reading data from file
- Writing data into file
- Closing a file

Steps for processing a file

- Declare a file pointer
- open a file using fopen() function
- Process the file using suitable file functions.
- close the file using fclose() function.

Declaration of a file

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

Syntax

FILE ***fp**;

Opening a file - for creation and edit

The fopen() function is used to create a new file or to open an existing file.

General Syntax :

fp = fopen("fileopen","mode")

For Example:

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function fclose().

fclose(fptr); //fptr is the file pointer associated with file to be closed.

File Opening Modes

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode

wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

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 exists 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.

Formatted File I/O Functions

Syntax of fprintf is

fprintf (fp, "control string", list);

Example: fprintf(fp1, "%s %d", name, age);

Syntax of fscanf is,

fscanf(fp, "control string", list);

Example: fscanf(fp, "%s %d", name, & age);

Note:

- fscanf is used to read list of items from a file
- fprintf is used to write a list of items to a file.

Note:

EOF – End of file (when EOF encountered the reading / writing should be terminated)

Example:

```
#include <stdio.h>
  main(){
    FILE *fp;
    fp = fopen("file.txt", "w");//opening file
   fprintf(fp, "Hello file by fprintf...\n");//writing data into file
    fclose(fp);//closing file
  }
```

Example 1: Write to a text file using fprintf()

```
#include <stdio.h>
int main()
{
  int num;
  FILE *fptr;
  fptr = fopen("C:\\program.txt","w");
  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
  printf("Enter num: ");
  scanf("%d",&num);
  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

Example 2: Read from a text file using fscanf()

```
#include <stdio.h>
int main()
{
   int num;
   FILE *fptr;
   if ((fptr = fopen("C:\\program.txt","r")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   fscanf(fptr,"%d", &num);
   printf("Value of n=%d", num);
   fclose(fptr);
     return 0;
}
```

Input/Output Operation on files

To perform Input/Output Operation on files we need below functions.

S.No	Function	Operation	Syntax
1	getc()	Read a character from a file	getc(fp)
2	putc()	Write a character in file	putc(c, fp)

3	fprintf()	To write set of data in file	fprintf(fp, "control string", list)
4	fscanf()	To read set of data from file.	fscanf(fp, "control string", list)
5	getw()	To read an integer from a file.	getw(fp)
6	putw()	To write an integer in file.	putw(integer, fp)

Unformatted File I/O Functions

fputc() function

The fputc() function is used to write a single character into file.

putc ( ):-Putting a character in to the file. It works with only character data type. One character at a time can write into a file.

Ex: char ch =’a’;

putc (ch, fp);

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file1.txt", "w");//opening file
  fputc('a',fp);//writing single character into file
  fclose(fp);//closing file
}
```

file1.txt

```
a
```

fgetc() function

The fgetc() function returns/read a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

getc ( ): getting a character from the file, or reading the file information character by character at a time, upto the end of the file by using this function.

Ex: char ch;

ch = getc (fp);

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("myfile.txt","r");
 while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

myfile.txt

this is simple text message

fputs()

The fputs() function writes a line of characters into file

Example:

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

```
#include<conio.h>
void main(){
FILE *fp;
clrscr();
fp=fopen("myfile2.txt","w");
fputs("hello c programming",fp);
 fclose(fp);
getch();
}
```

myfile2.txt

hello c programming

fgets()

The fgets() function reads a line of characters from file.

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
clrscr();
 fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));

fclose(fp);
```

```
    getch();
    }
```

Output:

hello c programming

The getw and putw functions:

These are integer oriented functions. These are similar to above functions and are used to read and write integer values. These are useful when we deal with only integer data. The general format is

putw ( ): putting or writing_of an integer_value to a file.

putw (integer , fp);

Ex:
```
int  x = 5;
putw(x,fp);
```

getw ( ): getting or reading integer value from a file.

Ex:
```
int x;
x = getw (fp);
```

File Positioning Functions

fseek()

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

syntax:

fseek(FILE * stream, long int offset, int whence)

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different Whence in fseek	
Whence	Meaning
SEKK_SET	Starts the offset from the beginning of the file.
SEKK_END	Starts the offset from the end of the file.
SEKK_CUR	Starts the offset from the current location of the cursor in the file.

or

fseek(file pointer, offset, position);

- *file pointer* is a pointer to the concerned file.
- *Offset* is a number or variable of type long, it specifies the number of positions (bytes) to be moved from the location specified. If offset is positive number, then moving forward or negative meaning move backwards.
- *Position* is a n integer number and it specifies from which position the file pointer to be moved. Position can take one of the following three values.

    0 beginning of file
    1 current position
    2 end of file

Eg: fseek (fp, 0L,0); - go to the beginning of the file. (Similar to rewind).
fseek (fp, 0L,1); - Stay at current position (Rarely used)
fseek (fp, 0L,2); -go to the end of the file, past the last character of the file.

Example:

```
#include <stdio.h>

 void main(){
```

```
    FILE *fp;
      fp = fopen("myfile.txt","w+");
    fputs("This is javatpoint", fp);
       fseek( fp, 7, SEEK_SET );
    fputs("sonoo jaiswal", fp);
    fclose(fp);
}
```

myfile.txt

This is sonoo jaiswal

rewind()

This function places the file pointer to the beginning of the file, irrespective of where it is present right now. It takes file pointer as an argument.

Syntax:

```
rewind( fp);
```

Example:

File: file.txt

```
this is a simple text
```

Example:

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

```
FILE *fp;
char c;
clrscr();
fp=fopen("file.txt","r");
while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
  rewind(fp);//moves the file pointer at beginning of the file
  while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
  fclose(fp);
getch();
}
```

Output:

this is a simple textthis is a simple text

As you can see, rewind() function moves the file pointer at beginning of the file that is why "this is simple text" is printed 2 times. If you don't call rewind() function, "this is simple text" will be printed only once.

ftell()

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

syntax:

n = ftell(fp);

n would give the relative offset(in bytes).

Example:

```
#include <stdio.h>
#include <conio.h>
void main (){
  FILE *fp;
  int length;
  clrscr();
  fp = fopen("file.txt", "r");
  fseek(fp, 0, SEEK_END);
    length = ftell(fp);
    fclose(fp);
  printf("Size of file: %d bytes", length);
  getch();
}
```

Output:

Size of file: 21 bytes

INBUILT FUNCTIONS FOR FILE HANDLING IN C LANGUAGE:

File handling functions	**Description**
fopen ()	fopen () function creates a new file or opens an existing file.
fclose ()	fclose () function closes an opened file.
getw ()	getw () function reads an integer from file.

putw ()	putw () functions writes an integer to file.
fgetc ()	fgetc () function reads a character from file.
fputc ()	fputc () functions write a character to file.
gets ()	gets () function reads line from keyboard.
puts ()	puts () function writes line to o/p screen.
fgets ()	fgets () function reads string from a file, one line at a time.
fputs ()	fputs () function writes string to a file.
feof ()	feof () function finds end of file.
fgetchar ()	fgetchar () function reads a character from keyboard.
fprintf ()	fprintf () function writes formatted data to a file.
fscanf ()	fscanf () function reads formatted data from a file.
fputchar ()	fputchar () function writes a character onto the output screen from keyboard input.
fseek ()	fseek () function moves file pointer position to given location.
SEEK_SET	SEEK_SET moves file pointer position to the beginning of the file.

SEEK_CUR	SEEK_CUR moves file pointer position to given location.
SEEK_END	SEEK_END moves file pointer position to the end of file.
ftell ()	ftell () function gives current position of file pointer.
rewind ()	rewind () function moves file pointer position to the beginning of the file.
getc ()	getc () function reads character from file.
getch ()	getch () function reads character from keyboard.
getche ()	getche () function reads character from keyboard and echoes to o/p screen.
getchar ()	getchar () function reads character from keyboard.
putc ()	putc () function writes a character to file.
putchar ()	putchar () function writes a character to screen.
printf ()	printf () function writes formatted data to screen.
sprinf ()	sprinf () function writes formatted output to string.
scanf ()	scanf () function reads formatted data from

	keyboard.
sscanf ()	sscanf () function Reads formatted input from a string.
remove ()	remove () function deletes a file.
fflush ()	fflush () function flushes a file.