UNIT -1

1: INTRODUCTION OF C LANGUAGE

INTRODUCTION OF COMPUTER LANGUAGES:

A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos[citation needed]. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description.

INTRODUCTION OF PROGRAMMING CONCEPT:

Computer programs are collections of instructions that tell a computer how to interact with the user, interact with the computer hardware and process data. The first programmable computers required the programmers to write explicit instructions to directly manipulate the hardware of the computer. This "machine language" was very tedious to write by hand since even simple tasks such as printing some output on the screen require 10 or 20 machine language commands. Machine language is often referred to as a "low level language" since the code directly manipulates the hardware of the computer.

By contrast, higher level languages such as "C", C++, Pascal, Cobol, Fortran, ADA and Java are called "compiled languages". In a compiled language, the programmer writes more general instructions and a compiler (a special piece of software) automatically translates these high level instructions into machine language. The machine language is then executed by the computer. A large portion of software in use today is programmed in this fashion.

INTRODUCTION OF C LANGUAGE:

C AS MIDDLE LEVEL LANGUAGE

All programming languages can be divided into two categories:

Problem oriented languages or High Languages

Examples of languages falling in this category are FORTRAN, BASIC, Pascal, etc. These languages have been designed to give a better programming efficiency, faster program development. Generally these languages have better programming capability but they are less capable to deal with hardware or Hardware related programming.

Machine oriented languages of Low Level Languages

Examples of languages falling in this category are Assembly language and Machine Language. These languages have been designed to give a better machine efficiency i.e. faster program execution. Generally these languages have better hardware programming capability but it is very difficult and tedious to do create complex application like and business application or some commercial application

C as Middle Level Languages:

C stands in between these two categories. That’s why it is called a Middle Level Language, since it was designed to have both; a relatively good programming efficiency (as compared to Machine Oriented Language) and relatively good machine efficiency (as compared to Problem Oriented Language).

History Of C language (Explian history of C language)

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A. **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**. It inherits many features of previous languages such as B and BCPL. Let's see the programming languages that were developed before C language.

Language	Year	Developed By	Remarks
Algol	1960	International Group	Too General, Too Abstract
BCPL	1967	Martin Richard	
B	1970	Ken Thompson	
Traditional C	1972	Dennis Ritchie	Modular,General,
K & R C	1978	Kernighan & Dennis Ritchie	
ANSI C	1989	ANSI Committee	
ANSI/ISO C	1990	ISO Committee	
C99	1999	Standardization Committee	

Overview of C

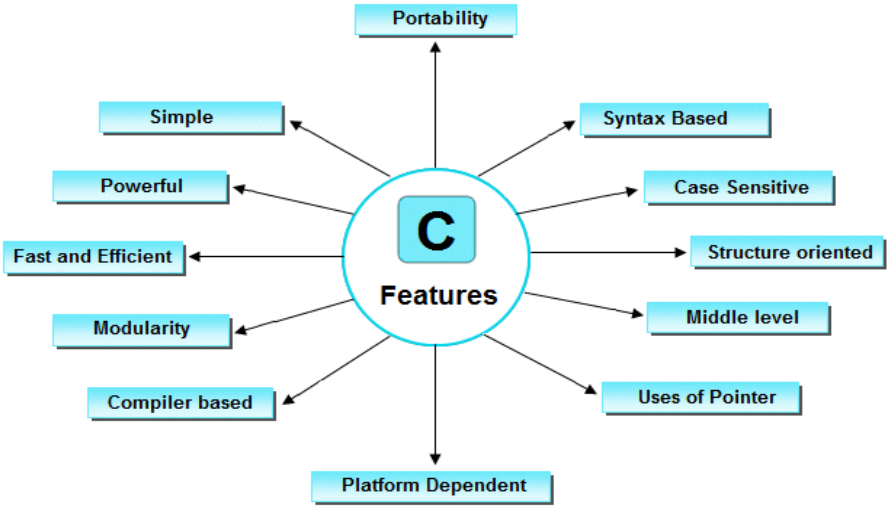
C is a computer programming language developed in 1972 by **Dennis M. Ritchie** at the Bell Telephone Laboratories to develop the UNIX Operating System. C is a simple and **structure oriented** programming language.

C is also called **mother Language** of all programming Language. It is the most widely use computer programming language, this language is used for develop system software and Operating System. All other programming languages were derived directly or indirectly from C programming concepts.

In the year 1988 'C' programming language standardized by ANSI (American national standard institute), that version is called **ANSI-C**. In the year of 2000 'C' Programming Language standardized by 'ISO' that version is called C-99

FEATURES OF C LANGUAGE: (Explain featuare of c language)

It is a very simple and easy language, C language is mainly used for develop desktop based application. All other programming languages were derived directly or indirectly from C programming concepts. This language has following features



Simple

Every c program can be written in simple English language so that it is very easy to understand and developed by programmer.

Platform dependent

A language is said to be platform dependent whenever the program is execute in the same operating system where that was developed and compiled but not run and execute on other operating system. C is platform dependent programming language.

Portability

It is the concept of carrying the instruction from one system to another system. In C Language.C file contain source code, we can edit also this code. .exe file contain application, only we can execute this file. When we write and compile any C program on window operating system that program easily run on other window based system.

When we can copy .exe file to any other computer which contain window operating system then it works properly, because the native code of application an operating system is same. But this exe file is not execute on other operation system.

Powerful

C is a very powerful programming language, it have a wide verity of data types, functions, control statements, decision making statements, etc.

Structure oriented

C is a Structure oriented programming language.Structure oriented programming language aimed on clarity of program, reduce the complexity of code, using this approach code is divided into sub-program/subroutines. These programming have rich control structure.

Modularity

It is concept of designing an application in subprogram that is procedure oriented approach. In c programming we can break our code in subprogram.

For example we can write a calculator programs in C language with divide our code in subprograms.

Example

```
void sum() {
 .....
 .....
}
void sub() {
 .....
 .....
}
```

Case sensitive

It is a case sensitive programming language. In C programming 'break and BREAK' both are different.

If any language treats lower case latter separately and upper case latter separately than they can be called as case sensitive programming language [Example c, c++, java, .net are sensitive programming languages.] other wise it is called as case insensitive programming language [Example HTML, SQL is case insensitive programming languages].

Middle level language

C programming language can supports two level programming instructions with the combination of low level and high level language that's why it is called middle level programming language.

Compiler based

C is a compiler based programming language that means without compilation no C program can be executed. First we need compiler to compile our program and then execute.

Syntax based language

C is a strongly tight syntax based programming language. If any language follow rules and regulation very strictly known as strongly tight syntax based language. Example C, C++, Java, .net etc. If any language not follow rules and regulation very strictly known as loosely tight syntax based language.

Efficient use of pointers

Pointers is a variable which hold the address of another variable, pointer directly direct access to memory address of any variable due to this performance of application is improve. In C language also concept of pointer are available.

CHARACTER SET

A character denotes any alphabet, digit, and special symbol used to represent information:

Alphabets: - A, B, C …Z- Upper Case, a, b, c……z - Lower Case

Digits: - 0, 1, … 9 - Decimal Digit

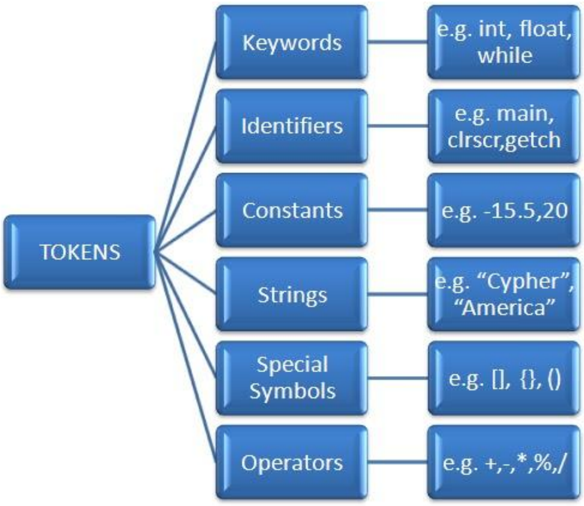
SPECIAL CHARACTERS:

,	Comma	**.**	Period	**;**	Semi Colon
:	Colon	**?**	Question Mark	**`**	apostrophe
"	Quotation Mark	**!**	Exclamation Mark	**|**	Vertical Bar
/	Slash	\	Back slash	**~**	Tilde
_	Under Score	**$**	Dollar sign	**%**	Per Cent Sign
#	Number Sign	**&**	Ampersand	**^**	caret
*****	Asterisk	**-**	Minus Sign	**+**	Plus
<	Opening angle bracket (less than sign)	**>**	Closing angle bracket (greater than sign)	**(**	Left parenthesis
)	Right parenthesis	**[**	Left bracket	**]**	Right bracket
{	Left brace / Open Corley bracket	**}**	Right brace / Close Corley bracket		

C Tokens (Explian C Tokens)

In a passage of text, individual words and punctuation marks are called tokens or lexical units. Similarly, the smallest individual unit in a c program is known as a token or a lexical unit. C tokens can be classified as follows:

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



KEY WORDS

Keywords are the words whose meaning has already been explained in the C compiler. The keywords cannot be used as a variable name because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the language. Some C compilers allow you to construct variable names which exactly resemble the keywords. However it would safer not to mix the variable names and the keywords. The keywords are also known as **'Reserve words'.** There are only 32 Keywords available in C. Following is the list of keywords available in C.

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

IDENTIFIER & VARIABLE

Identifiers are used as the general terminology for the names of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore (_) as a first character.

There are certain rules that should be followed while naming c identifiers:

- They must begin with a letter or underscore (_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only first 31 characters are significant.

Some examples of c identifiers:

Name	Remark
_A9	Valid
Temp.var	Invalid as it contains special character other than the underscore
void	Invalid as it is a keyword

What is a variable?

A variable is a meaningful name of data storage location in memory. When using a variable you refer to memory address of computer.

Naming Variables

Each variable has a name which is called variable name. The name of variable is also refered as identifier.

CONSTANTS

Constants in C represent fixed values that do not change during the execution of a program. Classification of 'C' constant is divided into several parts such as:

(A) Numeric Constant
1) Integer Constant
2) Real Constants

(B) Character Constant
1) Single Character Constants
2) String Constants

INTEGER CONSTANT

An integer constant refers to a sequence of digits. There are three types of integers decimal, octal and hexadecimal. An integer constant must have a prefix "int" before the declaring a constant i.e. int a. However you can assign number of constant in the same line like int a, b, c. There are several rules for integer constants given bellow:

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It could be either positive or negative.
4. If no sign precedes an integer constant it is assumed to be positive.
5. No commas or blanks are allowed within an integer constant.
6. The allowable range for integer constant is - 32768 to +32767.

Integer constants must fall within this range because the IBM compatible microcomputers are usually 16 bit computers that cannot support a number falling outside the above-mentioned range. Menace that it occupies 2(Two) bytes of memory. There are three types of integer constants:

❖ **Decimal Integer Constants**

Integer constants consisting of a set of digits, 0 through 9, preceded by an optional – or + sign.

Example of valid decimal integer constants 341, -341, 0, 8972

❖ **Octal Integer Constants**

Integer constants consisting of sequence of digits from the set 0 through 7 starting with 0 is said to be octal integer constants.

Example of valid octal integer constants 010, 0424, 0, 0540

❖ **Hexadecimal Integer Constants**

Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or 0X. They may also include alphabets from A to F representing numbers 10 to 15.

Example of valid hexadecimal integer constants 0xD, 0X8d, 0X, 0xbD

REAL CONSTANT

Integer numbers are inadequate to represent quantities that vary continuously. Such as distances, heights, temperatures, prices and so on. Numbers containing fractional parts like 17.548 represents these quantities. Such numbers are called real(or floating point/decimal point) constants. An float constant must have a prefix "float" before the declaring a constant i.e. float a. There are several rules for float constants given bellow:

1. A real constant must have at least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No commas or blanks are allowed within a real constant
6. Range of real constants expressed in exponential form is -3.4e38 to +3.4e38(10 digit and 6 decimal value after decimal point)

CHARACTER CONSTANT

The character type is used for storing characters such as letters and punctuation marks, but technically it is an integer type. How? Because the character type actually stores integers, not characters. To handle characters, the computer uses a numerical code in which certain integer represents certain characters. The most commonly used code is the ASCII code. There are two sub-division of character type.

❖ Single Character Constants

A single character constant contains a single character enclosed with a pair of single quote marks. Example of character constants is: ***'5', 'X', 'o', '+'*** **'%c'** is use to represent a single character constant. Hence, you use '%d' then it will print the ASCII value for the particular character like:

printf("%d",'a');

Would print 97, the ASCII value for character 'a'.

❖ Escape Characters/ Escape Sequences

C allows us to have certain non graphic characters in character constants. Non graphic characters are those characters that cannot be typed directly from keyboard, for example, tabs, carriage return, etc.

These non graphic characters can be represented by using escape sequences represented by a backslash (\) followed by one or more characters.

<u>**NOTE:**</u> An escape sequence consumes only one byte of space as it represents a single character.

Escape Sequence	Description
\a	Audible alert(bell)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\”	Double quotation mark
\’	Single quotation mark
\?	Question mark
\0	Null

String Constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank spaces.

<u>Example:</u>

"hello" , "1972" , "Good work" , "+-*/" , "2+a"

String constant can be declared using keyword char and length of the string followed by a square bracket. i.e. ***char add[25],*** in this case C will assign a string constant in memory it's variable name will be 'add' and length will be 25 character long.

Special Symbols

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

[] () {} , ; : * … = #

Braces {}: These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

Parentheses (): These special symbols are used to indicate function calls and function parameters.

Brackets []: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

OPERATORS:

C operators are symbols that trigger an action when applied to C variables and other objects. The data items on which operators act upon are called operands.

Depending on the number of operands that an operator can act upon, operators can be classified as follows:

1. Arithmetic Operator
2. Increment & Decrement Operator
3. Assignment Operator
4. Relational Operator
5. Logical Operator
6. Conditional Operator
7. Bitwise Operator
8. Special Operator
9. Sizeof Operator

Arithmetic Operators

Symbol	Description	Example (Suppose A=10 and B=3)	
+	Performs addition	A+B	13
–	Performs subtraction	A-B	7
*	Performs multiplication	A*B	30
/	Performs division	A/B	3
%	Performs modular division (Gives remainder after performing division)	A%B	1
+ (Unary Plus)	Gives value of operand	+A	10
- (Unary Minus)	Changes sign of its operand	-B	-3

Increment & Decrement Operators

C also provides the unary increment operator (++) and the unary decrement operator (--).

The increment operator adds 1 to its operand whereas the decrement operator subtracts 1 from its operand.

We can use increment and decrement operators in anyone of the following form:

Postfix: If increment and decrement operators are placed after a variable, they are said to be postfix increment or decrement operators. Postincrementing or postdecrementing the variable cause's current value to be used in expression in which it appears, and then variable value is incremented or decremented.

Prefix: If increment and decrement operators are placed before a variable, they are said to be prefix increment or decrement operators. Preincrementing or predecrementing the variable cause's current value to be incremented or decremented first and then only the variable value is used in the expression.

The rules can be summarized as:

- **Prefix: Change then use (++/-- variable)**
- **Postfix: Use then change (variable ++/--)**

Assignment Operators

Assignment operators are used to assign the result of an expression or a particular value to a variable.

The symbol of assignment operator is "=".

For example, a=11 will assign value 11 directly in a, and x=a+b will assign value of expression a+b in x. C also has a set of shorthand assignment operators. Consider this statement,

a=a+b;

The above statement can also be written using shorthand assignment operators as

a+=b;

Statement with simple assignment operator	Statement with shorthand assignment operator
a=a+1;	a+=1;
a=a-1;	a-=1;
a=a*(b+c);	a*=b+c;
a=a/(b+c);	a/=b+c;
a=a%b;	a%=b;

Relational Operators

All relational operators give answer in either TRUE or FALSE. Every non zero value is consider as TRUE by the compiler. Relational operators available in c are as follows:

Symbol	Meaning	Example
==	Comparison(equality)	2==2 (gives TRUE)
>	Greater than	5>6 (gives FALSE)
>=	Greater than or equal to	5>=5 (gives TRUE)
<	Less than	5<6 (gives TRUE)
<=	Less than or equal to	6<=5 (gives FALSE)
!=	Not equal to	5!=2 (gives TRUE)

Logical Operators

Unlike relational operators that establish relationship among values, logical operators refer to the ways these relationships among values can be connected.

There are three logical operators in c:

Symbol	Meaning	Example	Result
&&	AND (If anyone condition is False, answer is False)	6>=0 && 6>=20	False
||	OR (If anyone condition is True, answer is True)	2>=1 || 1>=5	True
!	NOT (It negates value of its operand and is an unary operator)	!(6<=6)	False

Conditional Operators

The only ternary operator available in c is "? :" to costruct conditional expression.

Syntax

expression_1? expression_2: expression_3;

The expression_1 is a logical or relational condition which is evaluated first. If the evaluated value is TRUE, then the value of complete expression will be expression_2 otherwise, it will be expression_3.

Bitwise Operators

There are also six bitwise operators available in c:

1. Bitwise AND Operator (&)
2. Bitwise OR Operator (|)
3. Bitwise Exclusive OR Operator (^)
4. Ones Complement Operator (~)
5. Left Shift Operator (<<)
6. Right Shift Operator (>>)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

```
#include <stdio.h>
main()
{
  unsigned int a = 60;   /* 60 = 0011 1100 */
  unsigned int b = 13;   /* 13 = 0000 1101 */
  int c = 0;
  c = a & b;      /* 12 = 0000 1100 */
  printf("Line 1 - Value of c is %d\n", c );
  c = a | b;      /* 61 = 0011 1101 */
  printf("Line 2 - Value of c is %d\n", c );
  c = a ^ b;      /* 49 = 0011 0001 */
  printf("Line 3 - Value of c is %d\n", c );
  c = ~a;         /*-61 = 1100 0011 */
  printf("Line 4 - Value of c is %d\n", c );
  c = a << 2;     /* 240 = 1111 0000 */
  printf("Line 5 - Value of c is %d\n", c );
  c = a >> 2;     /* 15 = 0000 1111 */
  printf("Line 6 - Value of c is %d\n", c );
}
```

```
Output
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Special Operators

Special operators available in c are as follows:

Comma Operator (,)

The comma operator is used to link the related expression together. Since, comma operator has the lowest precedence over other C operators; therefore it is always given in brackets. It is evaluated from left to right, and the evaluated rightmost value is the value of combined expression.

Example,

```
value=(a=10,a++,a*2);
```

In the above example, first 10 is assigned to a, then a is incremented by 1, and finally 11*2 is assigned to value variable.

Sizeof Operator

The sizeof operator is a compile time operator which returns the number of bytes that the operand occupies. The operand may be a variable, constant or a data type.

Syntax

Sizeof var;
or
Sizeof (type);

Where var is a declared variable and type is a valid c data type.

For example,

```
a=sizeof(sum);
b=sizeof(long int);
c=sizeof sum;
```

HIERARCHY OF OPERATION:

Precedence is a priority of the operator for evaluation or execution when there is more then 1 operator in a single expression or statement. If there are operators of same precedence then the expression will evaluated either from LEFT-TO-RIGHT. or RIGHT-TO-LEFT. This is called an associatively of an operator. There are two distinct priority levels of arithmetic operators in C.

Operator	Description	Associativity	Rank
( ) **[ ]** **.** **->** **++ --**	Parentheses (function call) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right	**1**
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of type) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left	**2**
*** / %**	Multiplication/division/modulus	left-to-right	**3**
+ -	Addition/subtraction	left-to-right	**4**
<< >>	Bitwise shift left, Bitwise shift right	left-to-right	**5**
< <=	Relational less than/less than or equal to	left-to-right	**6**

> >=	Relational greater than/greater than or equal to		
== !=	Relational is equal to/is not equal to	left-to-right	**7**
&	Bitwise AND	left-to-right	**8**
^	Bitwise exclusive OR	left-to-right	**9**
|	Bitwise inclusive OR	left-to-right	**10**
&&	Logical AND	left-to-right	**11**
| |	Logical OR	left-to-right	**12**
? :	Ternary conditional	right-to-left	**13**
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left	**14**
,	Comma (separate expressions)	left-to-right	**15**

Consider following expression

X=a-b/3 +c2-1 When a=9,b=12 and c=3 the statement becomes
X=9-12/3+3*2-1 and is evaluated as follows
First pass **Step-1 x=9-4+3*2-1**
Step-2 x=9-4+6-1
second pass **Step-3 x=5+6-1**
Step-4 x=11-1
Step-5 x=10

However the order of evaluation can be changed by introducing parentheses into an expression.

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

Answer will be x=7

Whenever parentheses are used, the expression within parentheses assumes higher priority. If there is more than one set of parentheses the operation within the inner most parentheses will be performed first followed by the operations within the second inner most and so on.

TYPE CASTING

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

(type_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation –

```
#include <stdio.h>
main() {
        int sum = 17, count = 5;
        double mean;
        mean = (double) sum / count;
        printf("Value of mean : %f\n", mean );
}
```

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

Value of mean: 3.400000

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value.

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. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

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 );
}
```

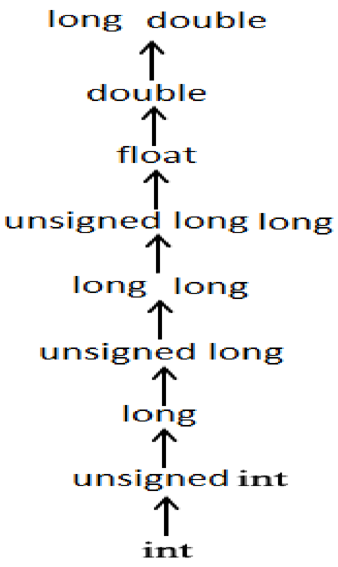
When the above code is compiled and executed, it produces the following result –

```
Value of sum : 116
```

Here, t0he 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 –



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept.

```
#include <stdio.h>
main() {
        int  i = 17;
        char c = 'c'; /* ascii value is 99 */
        float sum;
        sum = i + c;
        printf("Value of sum : %f\n", sum );
}
```

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

```
Value of sum: 116.000000
```

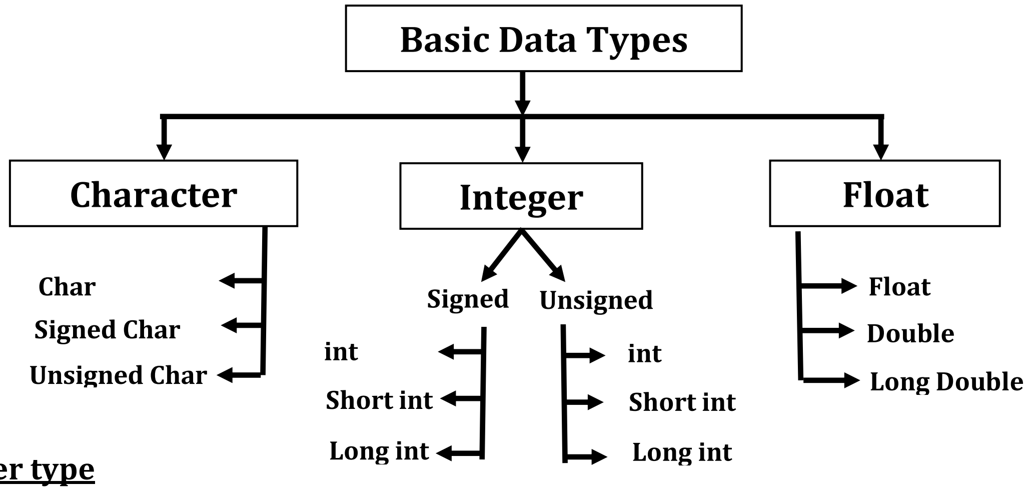
Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

DATA TYPES IN C : (Explain Data tyes in C)

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we use in our program. These data types have different storage capacities. The types in C can be classified as follows

S.N.	Types & Description
1	**Primary Data Type (Basic Types)** They are arithmetic types and are further classified into: (a) Character Types (b) Integer types and (c) Floating-point types
2	**The void Type** The type specifier **void** indicates that no value is available.
3	**User Define Data Type** **1. Type definition** By using a feature known as **"type definition"** that allows user to define an identifier that would represent a data type using an existing data type. **2. Enumerated Types** They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program
4	**Derived Types** They include (a) Pointer (b) Array (c) Structure and (d) Union

1. Primary Data Type (Basic Types)



Character type

Character types are used to store characters value. Size and range of Integer type on 32-bit machine. The following table provides the details of standard character types with their storage sizes and value ranges

Type	Size(bytes)	Range	Format (% code)
char or signed char	**1**	-128 to 127	%c
unsigned char	**1**	0 to 255	%c

Declare variable like - **char ans;**

Assigning value to character type variable - ans**='y';**

8	7	6	5	4	3	2	1
sign							
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Above given is a general structure of character variable. The leftmost bit is used to indicate the positive or negative value of variable. So the range for character variable is -2^7 to $2^{7-}1$. Means -128 to 127.

Integer type

Integers are used to store whole numbers. Size and range of Integer type on 32-bit machine. The following table provides the details of standard integer types with their storage sizes and value ranges

Type	Size(bytes)	Range	Format (% code)
short int / signed short int	1	-128 to 127	%d
int / signed int	2	-32,768 to 32767	%d
unsigned int	2	0 to 65535	%u
long / signed long	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu

Declare variable like

int rollno;
int accno;

To assign or to store integer value in this variable:

rollno=10;
accno=100;

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
sign															
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

Above given is a general structure of integer variable. The leftmost bit is used to indicate the positive or negative value of a variable. So the range for integer variable is -2^{15} to $2^{15}-1$. Means – 32768 to 32767.

Floating type

Floating types are used to store real (decimal points value) numbers. Size and range of Integer type on 32-bit machine. The following table provides the details of standard floating-point types with storage sizes and value ranges and their precisions.

Type	Size (bytes)	Range	Precision	Format (% code)
Float	4	3.4E-38 to 3.4E+38	6 decimal places	%f
double	8	1.7E-308 to 1.7E+308	15 decimal places	%lf
long double	10	3.4E-4932 to 1.1E+4932	19 decimal places	%Lf

Declare variable like

Float per;
Float rate;

To assign or to store some real value

Per=70.20;
Rate=123.45

32																															1
S I G n																															
2^{31}																															2^{0}

Above given is a general structure of float variable. The leftmost bit is used to indicate the positive or negative value of a variable. So the range for float variable is -2^{31} to $2^{31}-1$ Means -3.4e38 to +3.4e38

2. The void Type

The void type specifies that no value is available. It is used in three kinds of situations

A. Function returns as void

There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status)

;

B. Function arguments as void

There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, int getValue(void);

C. Pointers to void

A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function void *malloc(size_t size); returns a pointer to void which can be casted to any data type.

3. User define Data type

I. Type Definition

By using a feature known as **"type definition"** that allows user to define an identifier that would represent a data type using an existing data type.

General form: typedef type identifier ;

typedef existing_data_type new_user_define_data_type;

Example

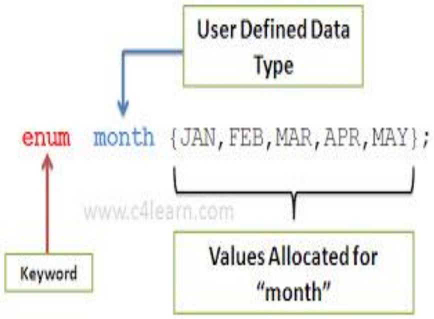
typedef int number;
typedef long big_ number;
	number visitors = 25;
	big_number population = 12500000;

II. Enumerated Type :- Enum

Syntex: enum identifier {value1, value2,.... Value n};

enum is **" Enumerated Data Type ".**

- enum is **user defined** data type
- In the above example **"identifier"** is nothing but the **user defined data type**.
- Value1, Value2, Value3..... etc creates **one set of enum values**.
- Using **"identifier"** we are **creating our variables**.


User Defined Data Type
enum month {JAN,FEB,MAR,APR,MAY};
www.c4learn.com
Keyword
Values Allocated for "month"

Example :

enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};

enum month rmonth;

1. First Line Create "User Defined Data Types" called month
2. It has 12 values as given in the pair of braces
3. In the second line "rmonth" variable is declare of "month" which can be initialized with any data values amongst 12 values rmonth = FEB;
   I. Default Numeric value assigned to first enum value is 0
   II. Numerically JAN is given value "0"
   III. FEB is given value "1"
   IV. MAR given values "2"

V. And so on…

```
printf("%d",rmonth);
```

It will print 1 on the screen because "Numerical Equivalent" of "FEB" is 1 and rmonth is initialized by FEB. Generally printing Values of enum variable is as good as **printing "integer"**

```
#include< stdio.h>
void main () {
        int i;
        enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};
        clrscr();
        for(i=JAN;i<=DEC;i++)
                printf("\n%d",i);
        getch(); }
```

4. Derived types

Those data types which are derived from fundamental data types are called derived data types. There are basically three derived data types.

1. **Array**: A finite collection of data of same types or homogenous data type.
2. **String**: An array of character type.
3. **Structure**: A collection of related variables of the same or different data types.
4. **Union** : Union is used to store collection of different kinds of data, just like a structure, however with union, you can only store information in one field at any one time
5. **Pointer:** A pointer is a variable whose value is the address of a memory location.

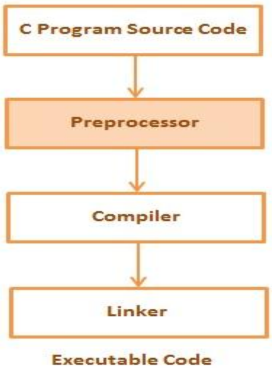
Pre Processor in C

The C Preprocessor is not part of the compiler but it extends the power of C programming language. All preprocessor directives begin with a **# symbol.**

The preprocessor step comes before compilation of source code and it instruct the compiler to do require pre-processing before actual compilation.

Types of Preprocessor Directives

- Macro Substitution
- Conditional Compilation
- File Inclusive



Below is the list of Preprocessor Directives

Directive	Description
#include	It includes header file inside a C Program.
#define	It is substitution macro. It substitutes a constant with an expression.
#if	It includes a block of code depending upon the result of conditional expression.
#else	It is complement of #if
#elif	#else and #if in one statement. It is similar to else if ladder.
#endif	It flags the end of conditional directives like #if, #elif etc.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true If constant is defined earlier using #define.
#ifndef	Returns true If constant is not defined earlier using #define.
#pragma	Issues special commands to the compiler.
#error	Prints error message on stderr.

#include Preprocessor Directives

#include Preprocessor Directives is used to include header file inside C Program. It checks for header file in current directory, If path is not mentioned. To include user defined header file we use double quote instead of using triangular bracket.

For Example

```
#include                                // Standard Header File
#include "myHeaderFile.h"               // User Defined Header File
```

First line tells the preprocessor to replace this line with content of string.h header file. Second line tells the preprocessor to get myHeaderFile.h from the current directory and add the content of myHeaderFile.h file.

#define Preprocessor Directives

It is simple substitution macro. It substitutes all occurrences of the constant and replaces them with an expression.

#define identifier value

#define: It is preprocessor directive used for text substitution.
Identifier: It is an identifier used in program which will be replaced by value.
Value: This is the value to be substituted for identifier.
Example

```
    #define PIE 3.141
    #define ZERO 0
```

#define macro substitution with arguments

#define Preprocessing directive can be used to write macro definitions with parameters. Whenever a macro identifier is encountered, the arguments are substituted by the actual arguments from the c program. No data type defined for macro arguments. You can pass any numeric like int, float etc.

Argument macro is not case sensitive.

Example

```
    #define circumference(r) (2*3.141*(r))
```

#if, #else and #endif Conditional Compilation Preprocessor Directives

The Conditional Compilation Directives allow us to include a block of code based on the result of conditional expression.

```
#if Condition_Expression
   statements;
#else
   statements;
#endif
```

It is similar to if else condition but before compilation. Condition_Expression must be only constant expression.

Predefined Macros in C Language

C Programming language defines a number of macros. Below is the list of some commonly used macros.

Macro	Description
NULL	Value of a null pointer constant.
EXIT_SUCCESS	Value for the exit function to return in case of successful completion of program.
EXIT_FAILURE	Value for the exit function to return in case of program termination due to failure.
RAND_MAX	Maximum value returned by the rand function.
__FILE__	Contains the current filename as a string.
__LINE__	Contains the current line number as a integer constant.
__DATE__	Contains current date in "MMM DD YYYY" format.
__TIME__	Contains current time in "HH:MM:SS" format.

#pragma Preprocessor Directive

The #pragma directive is used to give the preprocessor (and compiler) specific details on exactly how to compile the program. For example, specific compiler warnings can be ignored or warning levels changed up or down for different sections of code.

Example

```
#pragma page( ) // Forces a form feed in the listing
```

UNIT –1

2: INTRODUCTION OF LOGIC DEVELOPMENT TOOLS

IMPORTANCE OF PREPROGRAMMING TECHNIQUES:

Computer can solve variety of problems from the easiest to the most complex ones. To solve a problem it needs to be given a complete set of instructions. These instructions tell the computer what is to be done at every step. Remember one thing, computer does not solve a problem; it merely assists in solving the problem. To solve any problem we need to follow steps:

1. Define the problem.
2. Identify the input, output and constraint of the problem.
3. Identify different alternative of solving the problem.
4. Choose the best possible alternative from the above list.
5. Prepare detailed stepwise instruction set of the identified alternative.
6. Compute results using this instruction set.
7. Check correctness of the answer obtained.

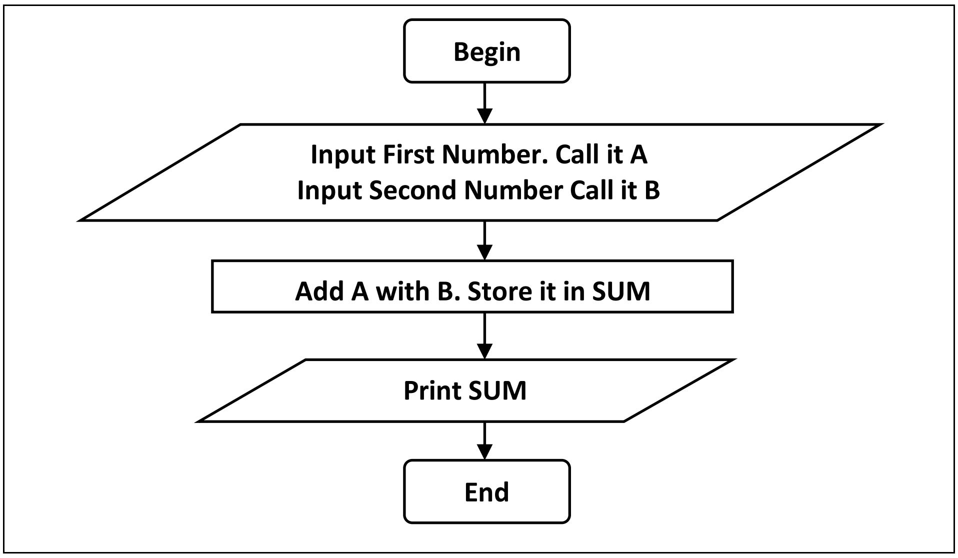
Steps 1 to 5 are performed by us, and computer comes into picture only in step 6 and 7. This simply suggests that the hard part is done by us. Programmer use different tools to help them develop program faster and without trouble. There mainly tools used for pre - programming are flowchart and algorithm.

PRE PROGRAMMING TOOLS:

FLOWCHART:

Flowchart is a graphical representation of, sequence of operations to be performed for solving a given problem. It uses different shapes to represent different types of instruction. These shapes are connected with solid lines having arrows that show the flow of instructions. It is perhaps the best available tool for representing what the computer must do, in order to solve a problem.

EXAMPLE:

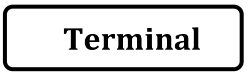
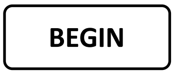
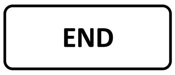
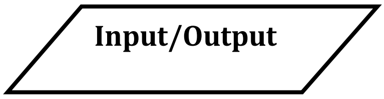
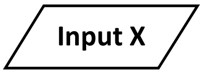
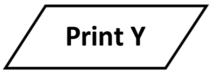
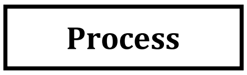
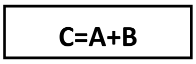
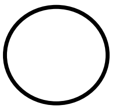
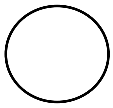
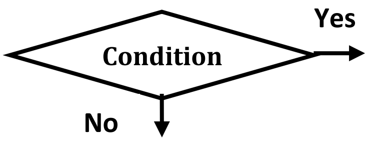
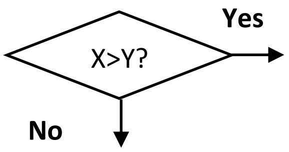
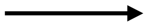
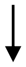

Begin
Input First Number. Call it A
Input Second Number Call it B
Add A with B. Store it in SUM
Print SUM
End

Above example shows the flow chart for adding two numbers. In this figure we have represented first number , second number and the addition value by the three terms namely “A”,”B”, “SUM”. These terms are known as variables.

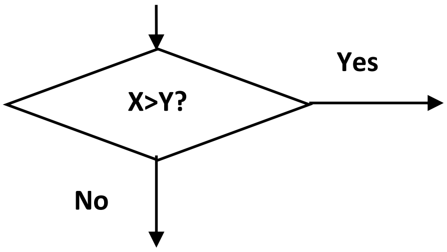
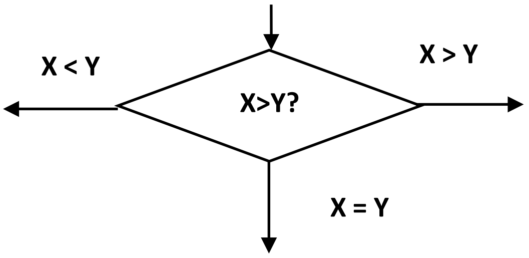
A variable is term used in programming language for the entity that has a tendency to change during the course of action. It can be defined as a symbol that is used in mathematical or logical expressions to represent a quantity that changes.

Component of flowchart:

American **N**ational **S**tandard **I**nstitute (**ANSI**) has standardized certain symbols that can be used in flowchart. These symbols can be broadly categorized as terminal, Input/Output, Processing, Decision, Connectors and Flow Line symbols.

NAME	SYMBOL	EXAMPLE	DESCRIPTION
Oval			Start Action Here
			Stop Action Here
Parallelogram			Take a value from an external source and assign it to X.
			Write a value of Y on Terminal / Screen.
Rectangle			Add the value of A and B, Place the result in C.
Circle	 Connecter		An entry in the flow chart is made at the connecting point marked as 1.
Diamond	 **Decision**		If X is greater than Y then, yes path is followed, otherwise no path is followed.
Flow Line			Shows the path to be followed.

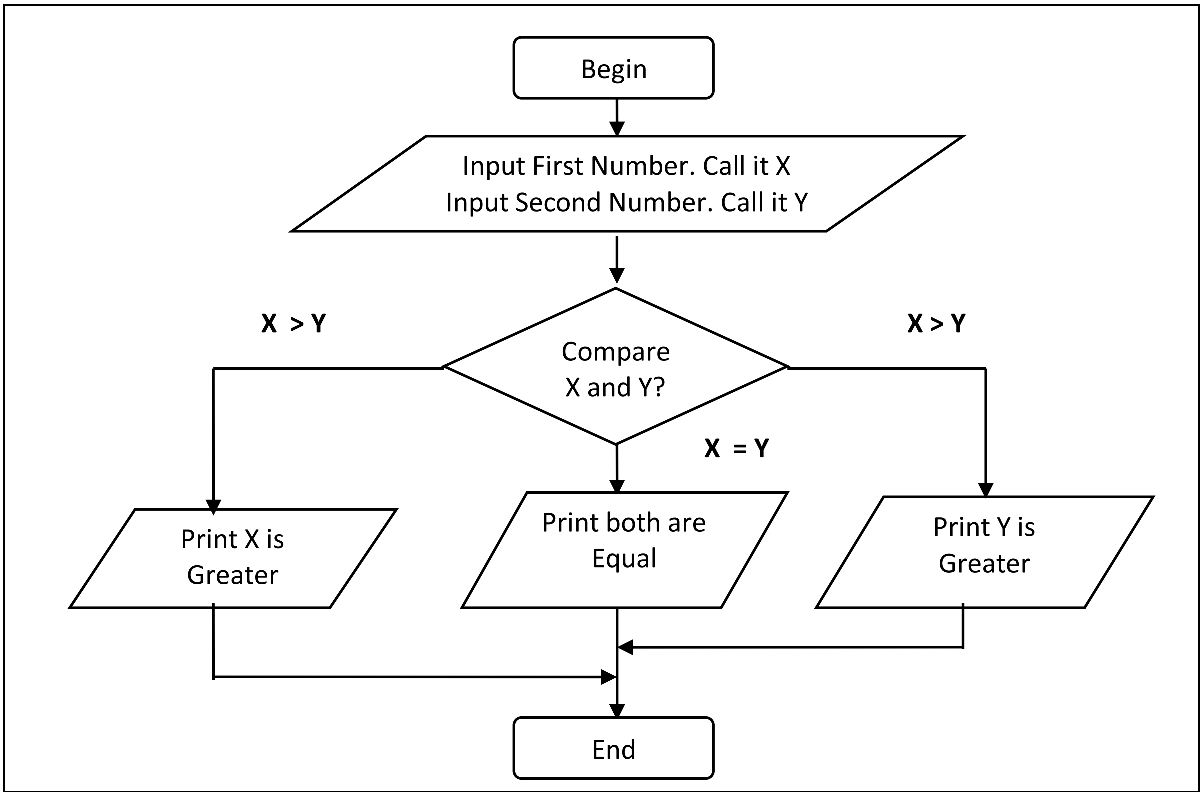
1. **<u>TERMINAL SYMBOL:</u>** As the name indicates these symbols are the first and the last symbol of a flowchart. Terminal symbol represent the starting and ending of the sequence of instruction in flowchart. This symbol is also used to indicate a pause in an instruction set.
2. **<u>INPUT/OUTPUT SYMBOL</u>**: This symbol is used to denote any functions input/output required in the instruction set. For example if there is an instruction that asks user to provide some value then this symbol will be used. Also if a user wants to print the result of an operation then again this symbol will be used.
3. **<u>PROCESSING SYMBOL</u>**: This symbol is used to represents the arithmetic and data movement operations required in the instruction set. Hence all arithmetic operations like addition subtraction, Division etc will be denoted using this symbol. We can represent series of sequential arithmetic operations using one processing symbol. The operations will be considered to be resolved in the way they appear.
4. **<u>DECISION SYMBOL:</u>** In the instruction set there may be a time where it becomes essential to skip certain steps and move on to some specific step. These kinds of situations are taken care by decision symbols. Here are some examples where we can use decision symbol.

Two way branch	Three way branch
	

5. **FLOW LINE:** To represent the flow of instruction in the set we use flow lines with arrowheads. It shows the sequence of instruction that is to be executed in a particular order. Intersection of the flow lines should be avoided whenever possible.
6. **Connecter symbol:** At times we may require more than a page for drawing a flowchart. In such situations it is necessary to see that the flow of logic does not go hay-wire. To prevent the flow lines going hay-wire and getting intersected at many places we use connecters.

EXAMPLE: Problem: Given two numbers X and Y find which one is greater.

Solution: The problem defined above can made be analogous to a real life example. Consider that you have been given a purse containing two coins. By performing following steps we can find which coin is having greater denomination.



1. Take out one coin from the purse and note its denomination.
2. Take out second coin from the purse and note its denomination.
3. Compare both the coins.
4. If denomination of both coins is same then, we have two equal coins.
5. If denomination of first coin is greater than it is the coin with greater denomination. Otherwise coin two has greater denomination.

Rules to draw a flowchart: As such there are no standard rules for designing a flowchart. But the rules mentioned below can be of great help in designing and representing a flowchart.

1. Think about the problems first; try to see what it requires as input, what it will give as output and what the constraints are.
2. Do not give all details in flowchart. Details can be specified in program.
3. The language used in symbol should be easy to understand.
4. Be consistent in using variable names in flowchart.
5. The sequence for drawing should be from left to right and top to bottom.
6. Avoid intersection of flow lines whenever possible by using connectors.
7. Make extensive use of connectors if the flow chart exceeds one page.

Advantage and Disadvantages of Flowchart:

Whatever tools we use they have some advantages. Flowchart is also not an exception. It also has some benefits and drawbacks.

ADVANTAGES OF FLOWCHART:

1. As it provides pictorial representation of the problem solution, it is easy to understand.
2. It provides a convenient aid for writing computer instructions.
3. It assists the programmer in reviewing and correcting the solution steps.
4. It helps in discussion of different methods for solving a problem. This in turn provides us a facility to compare them accurately.

DISADVANTAGES OF FLOWCHART:

1. Drawing a flowchart for a large and complex problem is time consuming and laborious.
2. As flowchart consists of symbols, any changes or modification in the program logic will most of the times require a new flowchart to be drawn.
3. There are no standard specifying the details of data that should be shown in flowchart. Hence flowchart for a given problem although with similar logic may vary in terms of data shown.

ALGORITHMS:

Development of an algorithm is popular approach to problem solving. This approach can also be used in the early stages of formulating computer solutions. We shall now focus on the concept of an algorithm. Let us first understand the meaning of the term algorithm.

WHAT IS AN ALGORITHM?

An algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step of the procedure. An algorithm includes calculations, reasoning and data processing. Algorithms can be presented by natural languages, pseudo code etc.

An algorithm is a procedure for solving problems to write a logical step-by-step method to solve the problem is called algorithm.

In other words, we can say that,

1. Step by step procedure for solving any problem is known as algorithm.
2. An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
3. An algorithm is a sequence of computational steps that transform the input into a valuable or required output.
4. Any special method of solving a certain kind of problem is known as algorithm

All Algorithms must satisfy the following criteria

1. Input: There are more quantities that are extremely supplied.
2. Output: At least one quantity is produced.
3. Definiteness: Each instruction of the algorithm should be clear and unambiguous.
4. Finiteness: The process should be terminated after a finite number of steps.
5. Effectiveness: Every instruction must be basic enough to be carried out theoretically or by using paper and pencil.

Properties of Algorithm

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm.

1. **Non Ambiguity:** Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in any algorithm should not denote any conflicting meaning. This property also indicates the effectiveness of algorithm.
2. **Range of Input:** The range of input should be specified. This is because normally the algorithm is input driven and if the range of input is not being specified then algorithm can go in an infinite state.
3. **Multiplicity:** The same algorithm can be represented into several different ways. That means we can write in simple English the sequence of instruction or we can write it in form of pseudo code. Similarly, for solving the same problem we can write several different algorithms.
4. **Speed:** The algorithmis written using some specified ideas. Bus such algorithm should be efficient and should produce the output with fast speed.
5. **Finiteness:** The algorithm should be finite. That means after performing required operations it should be terminate.

IMPORTANT OF CONCEPTS AND NOTATIONS:

We normally express our algorithms using a pseudo code that uses an informal language and some notation. A pseudo code is an outline of a program, written in a form that can easily be converted to a computer program. It allows the designer to focus on the logic of the algorithm rather than the details of language syntax. Computer science textbooks often use pseudo code in their examples, so that all programmers can understand them, even if they do not know the same programming language.

Usually, the pseudo code uses the concept of a program variable. It indicates a named location in the memory of a computer, in which a value can be placed. This value may then be used for any subsequent computation. Values to be stored in variables may be obtained from an input device such as keyboard of a computer, or by a computation using the value of previously stored variables. We may use names of our choice for variables. To place some value in a location associated with any variable, we shall use the back arrow (i.e. ←). For example, the following statement indicates that a value 5 is stored in the variable N.

N ← 5

The symbol ← used in the above statement is called an assignment operator.
Now consider the following tow statements

COUNT ← 3
COUNT ← COUNT + 1

The above statement stores the value 3 in the variable COUNT. The next statement first performs the computation indicated on the right hand side, and then stores the result in

the variable specified on the left hand side. The result of computation for the expression on the right hand side is 3 + 1 = 4. Hence, finally the value of the variable COUNT must be 4.
Note that we used the symbol + to indicate the operation of addition in the above example and ← is assignment operator. We shall use symbol -, *, /, and ↑ to indicate the operations of subtraction, multiplication, division and power respectively, in our algorithms.

Let us now proceed further for describing some other notations and concepts to be used in our pseudo code. One of them refers to the facility used for reading values to be operated upon. We shall use a READ statement for obtaining values of variables from an input device such as a keyboard. The READ statement contains a list of variables separated by commas. The following READ statement, for example, may be used to input values of the variable I, J and K through keyboard.

```
        READ (I, J, K)
```

Likewise, we shall use a WRITE statement for displaying or printing the values of the variables. For example, the following statement will display the values of the variables VAR1 and VAR2.

```
    WRITE (VAR1, VAR2)
```

We may also display a message enclosed between double quotes using the WRITE statement. For example, the following statement displays the message "Test Message" on the computer screen.

```
    WRITE ("TEST MESSAGE")
```

Algorithm often use a construct called IF – THEN – ELSE. It makes a decision in which a choice is made between two alternatives courses of action. We shall use the following general form for writing the IF statement in out pseudo code;

```
    IF condition THEN
        Sequence 1
    ELSE
        Sequence 2
    END IF
```

Here Sequence 1 and Sequence 2 represent sequence of instructions to be executed. The ELSE keyword and Sequence 2 are optional. The following example tests the score. If the score is less then 35, then the statement displays the message "Failed", else it displays message "Passed".

```
        IF score < 35  THEN
            WRITE ("Failed")
        ELSE
            WRITE ("Passed")
        END IF
```

A loop indicates that some operation is to be repeated as long as some condition is true. Programmer use the term "loop" to denote repetition in algorithms. A particular kind of loop which involves continuing repetition while some condition holds is frequently called a while loop. We shall use the following notation to specify a while loop.

```
    WHILE condition
         Sequence(s)
    END WHILE
```

WHILE is a loop with a simple conditional test at its beginning. We may form a condition using the well known relational operators (>, <, =, ≥, ≤, ≠) with their meanings.
Consider below example:

```
I ← 5
    WHILE I<10
        WRITE ("New Iteration")
    END WHILE
```

```
WRITE ("Bye!")
```

In example first initializes the value of the variable I with 5. It will then test whether the condition I<10 is true or not. If the condition is true, it executes all the statements within the loop in the indicated order, else it comes out of the while loop. Initially, since 5<10, the flow of control goes outside the loop. The message "New Iteration" is displayed. The next statement I ← I + 1 cause a new value 6 to be placed in the variable I. Now, once again the loop test is performed. This time the condition is found true, because the new value of I is 6, which is less than 10. Once again statements within the while loop are executed. This means, once again the message "New Iteration" will be displayed. And the value of I is incremented by 1. Hence the new value of I is 7. This process is repeated until the condition becomes false. Therefore, the above example will display the message "New Iteration" 5 times, and then the message "Bye!" would be displayed.

EXAMPLE:

Consider the problem of finding out an average of marks obtained in five different subjects by some student in some examination. We might follow the following general algorithm for solving that problem:

1. Read the set of five marks.
2. Compute the total marks by summing up the marks obtained in five different subjects.
3. Compute the average marks by dividing the total by 5.
4. Display the average marks.

We may represent the above algorithm in the form of following pseudo code:

```
1. READ (M1,M2,M3,M4,M5)
2. TOTAL ← M1 + M2 + M3 + M4 + M5
3. AVG ← TOTAL / 5
4. WRITE (AVG)
```

in the above example pseudo code, we used the variables M1,M2,M3,M4 and M5 for storing marks obtained in five different subjects. Moreover, the variable TOTAL is used to store the TOTAL of marks obtained in the five subjects. The variable AVG is used to store the average marks.

WHAT IS A LANGUAGE?

Language is a collection of words and symbols, which can be used to perform certain task or activities and to establish a communication between person to person.

In the same manner computer languages are the collection of predefine key words which can be used to perform certain task and to communicate between to entities like between two machines or between human and computers or computers and others peripherals.

There are three different level of programming languages. They are

(1) Machine languages (low level)
(2) Assembly languages
(3) Procedure Oriented languages (high level)
(4) Fourth Generation languages (4 GLS)

(1) Machine Languages:-

Computers are made of No. of electronic components and they all are two – state electronic components means they understand only the 0–(pulse) and 1(non–pulse). Therefore the instruction given to the computer must be written using binary numbers. 1 and 0. Computers do not understand English or any other language but they only understand or respond to binary numbers (0 and 1). So each computer has its own Machine languages.

(2) Assembly Languages:-

it was very tedious to understand and re member 0's representing numerous data and instruction. So to resolve this problem mnemonics codes were developed. For example add is used as symbolic code to represent addition. SUB is used for subtraction. They are the symbolic representation of certain combination of binary numbers.

for example

SUB

Which represents certain actions as computer understand only Machine language instructions a program written in Assembly Language must be translated into Machine languages. Before it can be executed this translation if done by another program called assembler. This assembler will translate mnemonic codes into Machine languages.

(3) Procedure Oriented Languages :-

In earlier assembly languages assembler program s produced only one Machine languages instruction for every assembly languages instruction. So to resolve this problem now assembler was introduced. Which can produce several machine level instructions f or one assembly language instruction. Thus programmer was relieved from the task of writing and instruction for every machine operation performs ed. These languages contain set of words and symbols and one can write program with the combination of these keywords. These languages are also called high level languages. Basic, FORTRAN, Pascal and COBOL. The most important characteristic of high level languages is that it is machine independent and a program written in high level languages can be run on any computers with different architecture with no modification or very little modification.

WHAT IS LANGUAGE TRANSLATORS?

As we know that computer understands only the instruction written in the Machine language. Therefore a program written in any other language should be translated to Machine language. For this purpose special program s are available they are called translators or language processors? This special program accepts the user program and checks each statement and produces a corresponding set of Machine language instructions. There are two types of Translators.

(1) COMPILER: - A **Compiler** checks the entire program written by user and if it is free from error and mistakes then produces a complete program in Machine language known as object program. But if it founds some error in program then it does not execute the single statement of the program. So compiler translate whole program in Machine language before it starts execution.

(2) INTERPRETERS: - Interpreters performs the similar job like compiler but in different way. It translates (Interprets) one statement at a time and if it is error – free then executes that statement. This continues till the last statement in the program has been translated and executed. Thus Interpreter translates and executes the statement before it goes to next statement. When it founds some error in statement it will immediately stop the execution of the program. Since compiler of Interpreter can translate only a particular language for which it is designed one has to use different compiler for different languages.

Difference of Compiler – Interpreter

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

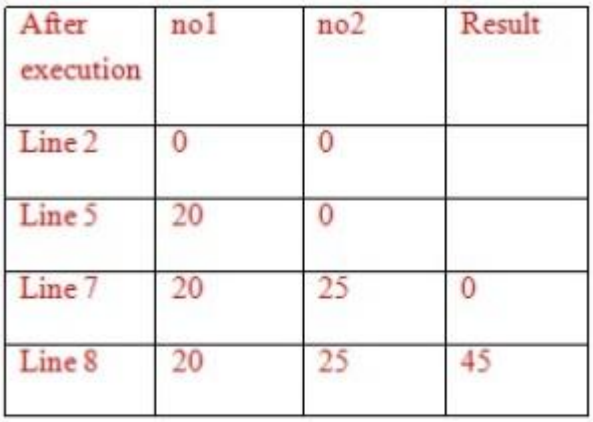
DRY RUN and Its USE

C Language Including Examples a dry run is a testing process where the effects of a possible failure are intentionally mitigated. With computer-programming, a dry run is a mental run of a Program, in which the computer coder examines the source signal one action at any given time in addition to determines what it will perform as soon as function. Use of dry run inside acclaim procedures is supposed because following: this manufacturing area (subcontractor) ought to perform a finish test in the system it offers to produce before the real acclaim from your company side.

You can attempt your current system without making use of a pc by simply dried managing the idea in some recoverable format. Anyone work as this computer – adopting the instructions in the system, documenting this valuation in the varying in just about every point. You can so this having a stand. The stand along with possess line on course with the names in the specifics from the system. Each and every strip from the stand will probably be classed having a range amount from your system.

Example regarding dried function regarding amount of two numbers:

```
L1 Declare two variables, n1, n2
L2 Initialize both values to 0
L3 n1=0, n2=0
L4 Ask the user to enter value for no1
L5 Assign user input to no1
L6 Ask the user to enter value for no2
L7 Assign user input to no2
L8 Add n1 to n2
L9 Print result
```

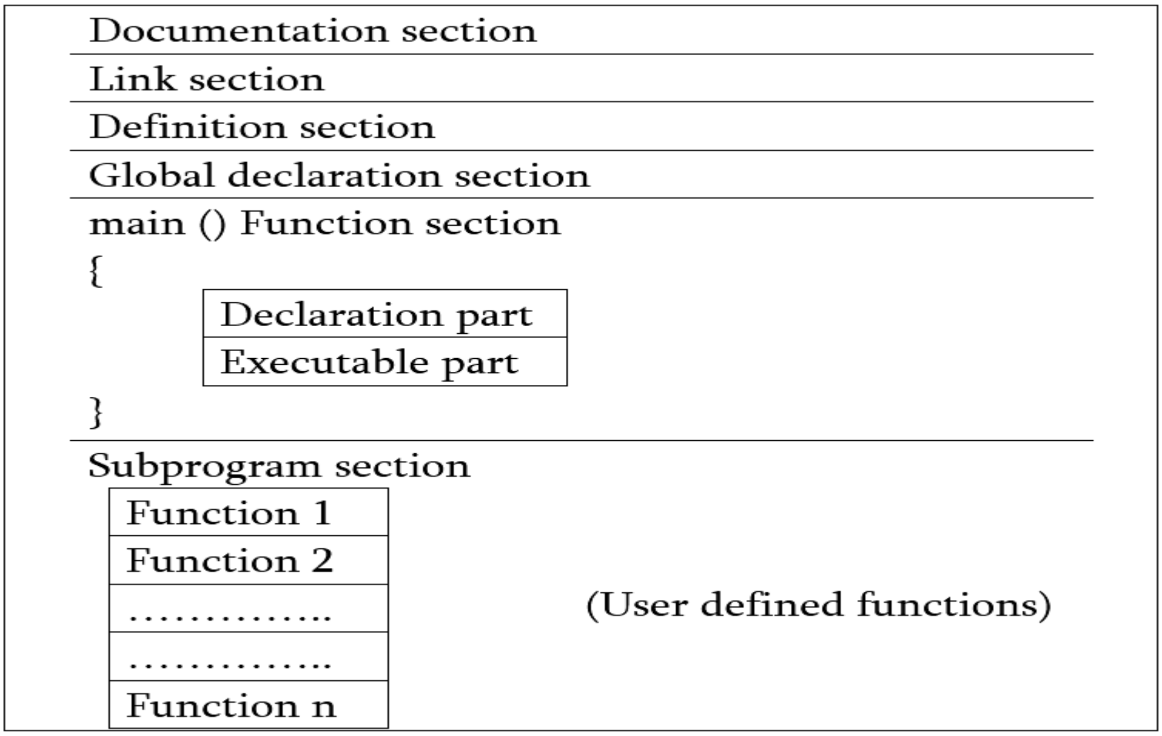


After execution	no1	no2	Result
Line 2	0	0	
Line 5	20	0	
Line 7	20	25	0
Line 8	20	25	45

C PROGRAM STRUCTURE:

C language is very popular language among all the languages. Sometimes I think that there had been no "C" language there would have no C++ and even Java. So let me explain you the basic structure of a C language.

The structure of a C program is a protocol (rules) to the programmer, while writing a C program. The general basic structure of C program is shown in the figure below. The whole program is controlled within main ( ) along with left brace denoted by "{" and right braces denoted by "}". If you need to declare local variables and executable program structures are enclosed within "{" and "}" is called the body of the main function. The main ( ) function can be preceded by documentation, preprocessor statements and global declarations.



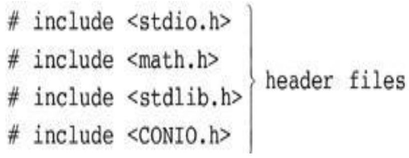
1. Documentation section:

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

SYMBOL	TYPE OF COMMENT
/* */	Multi Line Comment
//	Single Line Comment

2. Link section

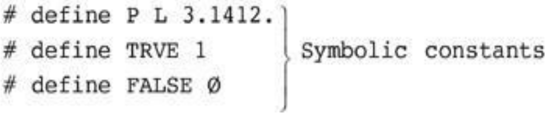
The link section provides instructions to the compiler to link functions from the system library such as using the #include directive.



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

3. Definition section

The definition section defines all symbolic constants such using the **#define** directive.



```
# define P L 3.1412.
# define TRVE 1
# define FALSE Ø
```

NOTE : Pre Processor statement (link section and Definition section)

4. Global declaration section

There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

5. main () function section

Every C program must have one main function section. This section contains two parts; declaration part and executable part

I. **Declaration part:** The declaration part declares all the variables used in the executable part.

II. **Executable part:** There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

6. Subprogram section

If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

UNIT -2
CONTROL STRUCTURES

Selctive Control Structure

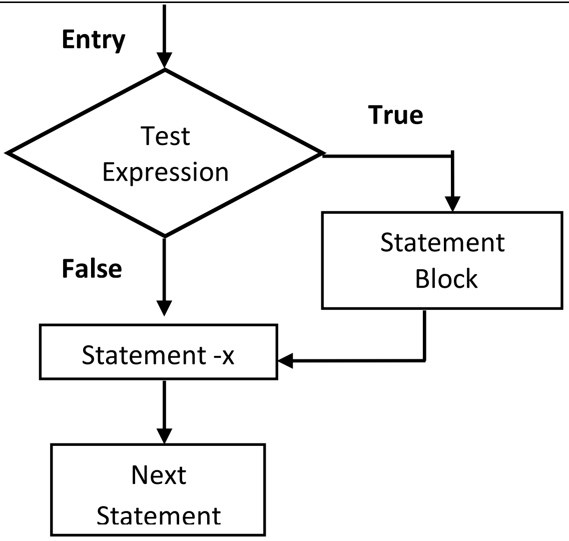
What is Decision making Statement?

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

If statement

if statement is a powerful decision making statement and is used to control the flow of execution of statements. It takes the following form.

SYNTAX	Flowchart

if (test expression/Condition)
{
Statement block;
}
Statement - x;
	 Entry
	Test Expression
	True
	Statement Block
	False
	Statement -x
	Next Statement

It allows the computer to evaluate the expression/condition first and then, depending on **whether the value of the expression/condition is true or false**, it transfers the control to a particular statement. This point of program has **two paths** to follow one for the true condition and other for the false condition. The statement block may be a single statement or a group of statements.

If the **test expression is true** the statement–block will be executed; otherwise the statement – block will be skipped and the **execution will jump to the statement–x**.

When the condition is true both the statement–block and the statement – x is executed in sequence.

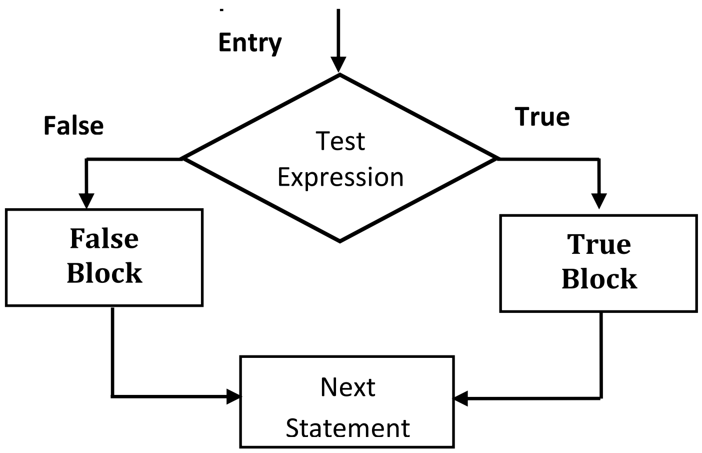
EXAMPLE:

```
#include <stdio.h>
int main()  {
   int number;
   printf("Enter an integer: ");
   scanf("%d", &number);
   if (number == 0) {
      printf("You entered Value is Zero ");
   }
       printf ("The if statement is easy.");
       return 0;
}
```

if - else statement

if – else statement is an extension of the simple if statement. The general form is as below.

SYNTAX	FLOWCHART

if (test expression)
{
Statement(s); - ***True block***
}
else
{
Statement(s); - False block
}
Statement Next;
	 Entry
	False
	Test Expression
	True
	False Block
	True Block
	Next Statement

It allows the computer to evaluate the expression/condition first and then, depending on **whether the value of the expression/condition is true or false**,

It transfers the control to a particular statement. This point of program has **two paths** to follow one for the **true condition** and other for the **false condition.** The statement block may be a single statement or a group of statements.

If the test expression is Ture, then the True block statement(s), immediately executed; otherwise the **False** block statement(s) are executed; and go (jump) to the next statement

In the both the case either true-block or false–block will be executed, not both block statement(s) is run.

EXAMPLE:

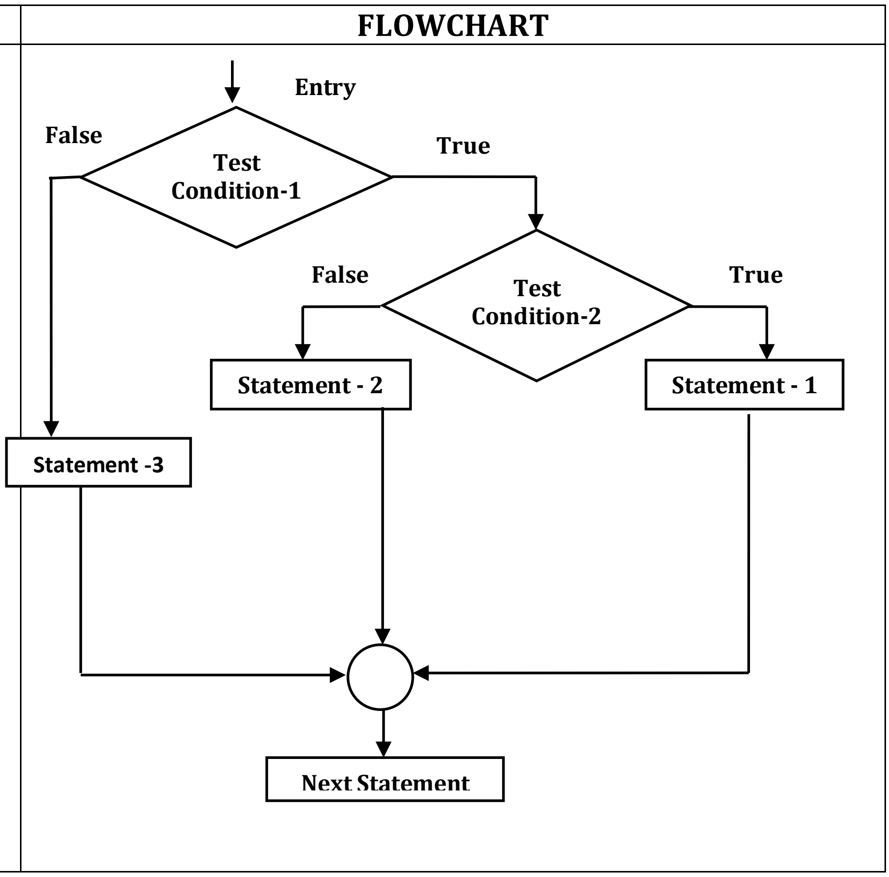
```
#include <stdio.h>
int main()
{
   int number;
   printf("Enter an integer: ");
   scanf("%d",&number);
   if( number%2 == 0 )  // True if remainder is 0
           printf("%d is an even integer.",number);
  else
           printf("%d is an odd integer.",number);
   return 0;
}
```

When user entered 7, the test expression (number%2 == 0) is evaluated to false because of reminder of 7%2 = 1. Hence, the statement inside the body of else statement printf ("%d is an odd integer"); is executed and the statement inside the body of if is skipped.

Nested if statement

When a series of decisions are involved we may have to use more then one if-else statements in nested form as follows.

SYNTAX	FLOWCHART

if (condition 1)
{
// If the test condition 1 is TRUE
// then it will check condition 2
if (condition 2)
{
// Both Condition are TRUE
// condition-1 and 2
Statement(S) - 1;
}
else
{
// Condition -1 is TRUE and
// Condition - 2 is False
Statement(s) - 2;
}
}
else
{
// Condition -1 is False
Statement(s) - 3;
}
	 Entry
	False
	Test Condition-1
	True
	False
	Test Condition-2
	True
	Statement - 2
	Statement - 1
	Statement -3
	Next Statement

SYNTAX:

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

EXAMPLE

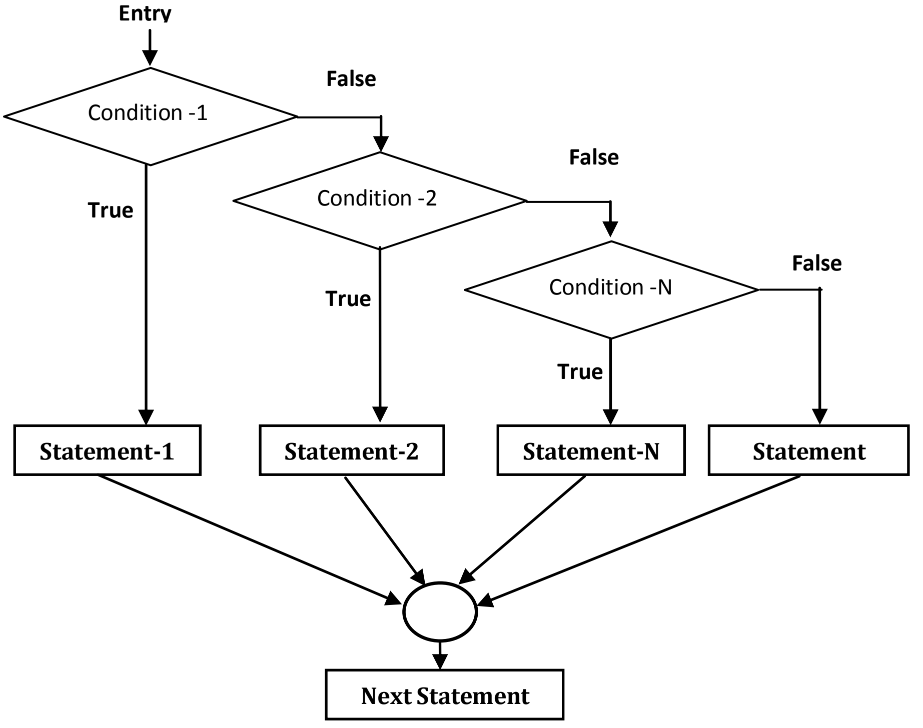
```
#include <stdio.h>
 int main() {
 int n;
 printf("Please Enter Number:");
 scanf("%d",&n);
 if ( n != 0 ) {
         if (n > 0 )     {
                 printf("\n Number is Positive");          Statement - 1
        }
        else    {
                 printf("\n Number is Negative");         Statement - 2
       } }
 else {
         printf("\n  Number is Zero.");                    Statement - 3
  }
  return 0;
}
```

If the user entered a number less than or greater than 0(Zero) then check condition-1 if the condition-1(n!=0) is True then again check condtion-2 (n>0); if the condition-2 become a True the statement-1 will be executed or if condition 2 become a False then statement-2 will be executed. If the condition-1(n! =0) is become a False then go to then Statement-3 then the control is transferred to the statement–x.

Else if **statement (Else If ladder)**

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

SYNTAX	FLOWCHART

**if (condition 1)
{
statement - 1;
}
else if (condtion 2)
{
statement - 2;
}
.
.
.
else if (condition N)
{
statement - n;
}
else
{
default statment;
}
statement-x;**
	 Entry
	Condition -1
	False
	True
	Condition -2
	False
	True
	Condition -N
	False
	True
	Statement-1
	Statement-2
	Statement-N
	Statement
	Next Statement

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

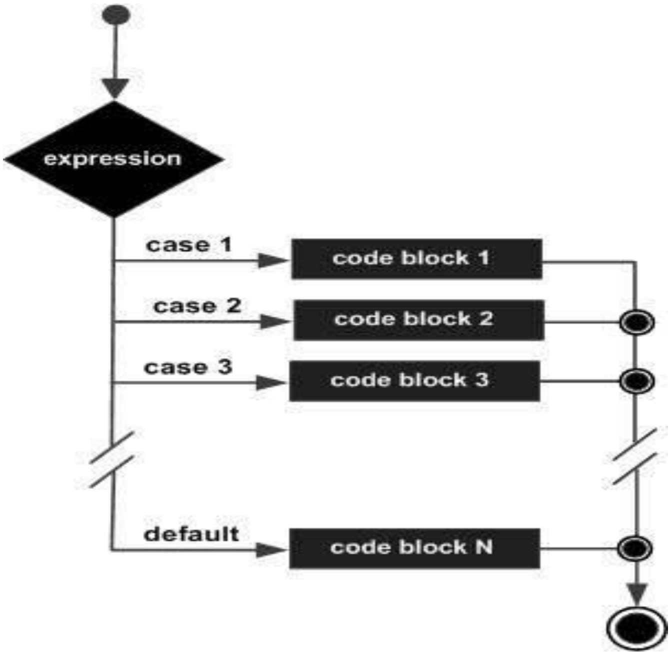
EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
 main()
{
         int marks;
        printf("Enter your marks between 0-100\n");
        scanf("%d", &marks);
        if(marks >= 90)                          // Marks between 90-100
        {
                printf ("YOUR GRADE : A\n");
        }
        else if (marks >= 70 && marks < 90)       // Marks between 70-89
        {
                printf("YOUR GRADE : B\n");
        }
        else if (marks >= 50 && marks < 70)       //  Marks between 50-69
        {
                printf("YOUR GRADE : C\n");
        }
        else                                      // Marks less than 50
        {
                printf("YOUR GRADE : Failed\n");
        }
        getch();
 }
```

SWITCH-CASE:

C has a built in multiway decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below.

SYNTAX	FLOWCHART

switch(expression / Variable)
{
case value 1:
block -1
break;
case value 2:
block -2
break;
___
___
default:
default –Block
}
Statement –x;
	 expression
	case 1
	code block 1
	case 2
	code block 2
	case 3
	code block 3
	default
	code block N

The expression is an integer expression or character value -1, value-2 is constants or constant expressions are known as case labels. Each of these values should be unique within a switch statement. Block-1, block-2 are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that labels end with colon. When a switch is executed, the value of the expression is successively compared against the values value-1,value-2……if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

Notice that each group of statement ends with a break statement in order to transfer control out of the switch statement. The break statement is required within each of the first groups in order to prevent the succeeding groups of statement from executing. This last group does not require a break statement since control will automatically be transferred out of the switch statement after the last group has been executed. This last break statement is included; however has a matter of good programming practice. So that it will be present, if another group of statement is added later. The selection process of switch statement is illustrated in the flow chart.

EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
 main()
{
            int num;
            printf("\nEnter Number between 1-3 : ");
            scanf("%d", &num);
            switch(num)
            {
                    case 1:
                           printf("I am in case 1 \n");
                           break;
                    case 2:
                           printf("I am in case 2 \n");
                           break;
                    case 3:
                           printf("I am in case 3 \n");
                           break;
                    default:
                           printf("I am in defult\n");
                           break;
            }
            getch();
}
```

CONDITIONAL (TERNARY) OPERATOR:

Simple conditional operations can be carried out with the conditional operator (?:) An expression that makes the use of the conditional operator is called a conditional expression. Conditional operators are also known as turnery operators.

SYNTAX: ***(condition)? True : False***

Exp1 ? Exp2 : Exp3

Here if condition is evaluated as true then value of exp2 will be return and if condition is evaluated as false then exp3 will be return. For example

EXAMPLE: T= (I<0)? 0: 100

If I variable value is less then 0 then condition is evaluated as true and T will be assigned with 0 but if I variable value is greater then 0 then condition is evaluated as false and 100 will be assign to variable T. For example

printf("%d", (I<0) ? 0:100);

Iterative (Looping) Control Statements

What is LOOP?

A loop is used for executing a block of statements repeatedly until a given condition returns false

An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.

- **Counter not Reached:** If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
- **Counter reached:** If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

Infinite loop:

The statements get executed many numbers of times based on the condition. But if the condition is given in such logic that the repetition continues any number of times with no fixed condition to stop looping those statements, then this type of looping is called ***infinite looping***.

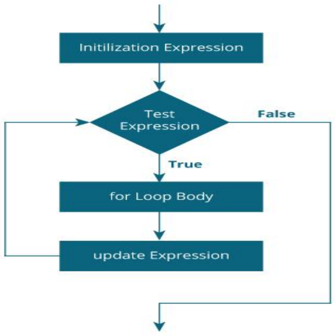
There are mainly two types of loops:

1. **Entry Controlled loops**: In this type loops, the test condition/expression is tested or evaluated before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
2. **Exit Controlled Loops**: In this type loops, the test condition/expression is tested or evaluated at the end of loop body. Therefore, the loop body will execute **atleast once**, irrespective of whether the test condition is true or false. **Do–while loop** is exit controlled loop.

FOR Loop :

A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

SYNTAX	FLOWCHART

for (initialization;test-expr;update-expr)
{
// body of the loop
// statements we want to execute
}
	 Initilization Expression
	Test Expression
	False
	True
	for Loop Body
	update Expression

A loop variable is used to control the loop. First initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If statement is true, then loop body is executed and loop variable gets updated. Steps are repeated till exit condition comes.

- **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int i=1;
- **Test Expression**: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;

- **Update Expression**: After executing loop body this expression increments / decrements the loop variable by some value. **for example: i++ , i--;**

EXAMPLE :

int i;
for(i=1; i<=10;i++)
printf("%d",i);
	int i;
	for(i=10; i>=1;i--)
	printf("%d",i);

Output 12345678910	Output 10987654321

That the initialization, testing and incrementation of loop counter is done in the for statement itself. Instead of i=i+1, the statement i++ or i+=1 can also be used. Since there is only one statement in the body of the for loop, the pair of braces have been dropped. The for statement allows for negative increments. This loop is also executed 10 times. But the output would be from 10 to 1 instead of 1 to 10. Note that, braces are optional when the body of the loop contains only one statement.
We can also write for loop this way.

int i;
for(i=1; i<=10;)
{
printf("%d",i);
i=i+1;
}
Output is:12345678910
	int i=1;
	for(; i<=10;)
	{
	printf("%d",i);
	i=i+1;
	}
	Output is:12345678910

Here the increment is done within the body of the for loop and not in the for statement. Note that inspite of this the semicolon after the condition is necessary. In another example neither initialization, nor the incrementation is done in the for statement, but still two semicolon are required.

Additional features of for loop : - Multiple initializations:

We can have multiple initialization in the for loop as shown below.

Example:

```
for (i=1,j=1;i<10 && j<10; i++, j++)
```

Difference between above for loop and a simple for loop

1. It is initializing two variables.
2. It has two test conditions joined together using AND (&&) logical operator.
   Note: You cannot use multiple test conditions separated by comma, you must use logical operator such as **&& or ||** to join conditions.
3. It has two variables in increment part.

Note: Should be separated by comma.

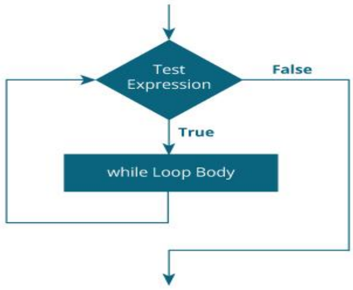
WHILE Loop :

A loop we have seen that the number of iterations is known beforehand, the number of times the loop body is needed to be executed is known to us.

While loops are used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of test condition.

We have already stated that a loop is mainly consisted of three statements- initialization, test expression, update expression. The syntax of the while

SYNTAX	FLOWCHART

Initialization;
while(test expression/condition)
{
Statement 1;
Statement 2;
increment/decrement;
}
	

EXAMPLE:

Suppose we want to display the consecutive digits 1,2,3,…10. This can be accomplished with the following program.

```
#include<stdio.h>
main()
{
   int i=1;
   while(i<=10)
   {
     printf("%d",i);
     i++;
    }
}
Output:12345678910
```

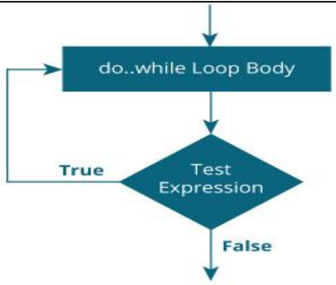
Initially, digit is assigned a value of 1. The while loop then displays the current value of digit, increases its value by 1 and then repeats the cycle, until the value of the loop will be repeated by 10 times, resulting in 10 consecutive numbers of output.

DO WHILE Loop :

In do while loop first the statements in the body are executed then the condition is checked. If the condition is true then once again statements in the body are executed. This process keeps repeating until the condition becomes false. As usual, if the body of do while loop contains only one statement, then braces ({}) can be omitted. Notice that unlike the while loop, in do while a semicolon (;) is placed after the condition.

The do while loop differs significantly from the while loop because in do while loop statements in the body are executed at least once even if the condition is false. In the case of while loop the condition is checked first and if it true only then the statements in the body of the loop are executed

SYNTAX	FLOWCHART

Initialization;
do
{
Statement 1;
Statement 2;
increment/decrement;
} while(test condition);
	

Example:

Suppose we want to display the consecutive digits 1,2,3,…10. This can be accomplished with the following program.

```
#include<stdio.h>
main()
{
   int i=1;
   do
   {
     printf("%d",i);
     i++;
    } while(i<=10);
}
Output:12345678910
```

The digit is initially assigned a value of 1. The do while loop displays the current value of digit. Increases its value by 1, and then tests to see if the current value of digit exceeds 10. If so loop terminates; otherwise the loop continues, using the new value of digit.

Entry Controlled Loop v/s Exit Controlled Loop

Entry Controlled Loop	Exit Controlled Loop
Test condition is checked first, and then loop body will be executed.	Loop body will be executed first, and then condition is checked.
If Test condition is false, loop body will not be executed.	If Test condition is false, loop body will be executed once.
For loop and while loop are the examples of Entry Controlled Loop.	do while loop is the example of Exit controlled loop.
Entry Controlled Loops are used when checking of test condition is mandatory before executing loop body.	Exit Controlled Loop is used when checking of test condition is mandatory after executing the loop body.

Nested loop in C

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

SYNTAX		
For loop	**While loop**	**Do… While Loop**

for(initialization;test-expr;update-expr)
{
// statements
for(initialization;test-expr;update-expr)
{
// Inner loop statements
}
// statements
}
	while(condition-1)
	{
	// statements
	while(condition-2)
	{
	// Inner loop statements
	}
	// statements
	}
		do
		{
		// statements
		do
		{
		// Inner loop statements
		}while(condition-2);
		// statements
		}while(condition-1);

```
#include <stdio.h>
void main()
{
        int i, j;                 /* Loop counter variable declaration */
        for(i=1; i<=10; i++)      /* Outer loop */
        {
            for(j=1; j<=5; j++)   /* Inner loop */
            {
                printf("%d\t", (i*j));
            }
                printf("\n");     /* Print a new line */
        }
        getch();
}
```

Let us take a note on above program.

To understand the above program easily let us first focus on inner loop.

- First the initialization part executes initializing j=1. After initialization it transfer program control to loop condition part i.e. j<=5.
- The loop condition checks if j<=5 then transfer program control to body of loop otherwise terminated the inner loop.
- Body of loop contains single printf("%d\t", (i*j)); statement. For each iteration it print the product of i and j.
- Next after loop body, the loop update part receives program control. It increment the value of j with 1 and transfer program control back to loop condition.

From the above description it is clear that the inner loop executes 5 times.

For i=1 it prints the product of i and j

```
1   2   3   4   5
```

Similarly for i=2 it prints

```
2   4   6   8   10
```

Next let us now concentrate on outer loop.

- In the outer loop first variable i is initialized with 1. Then it transfer program control to loop condition i.e. i<=10.
- The loop condition part checks if i<=10 then transfer program control to body of loop otherwise terminate from loop.
- Body of loop does not contain any statement rather it contain another loop i.e. the inner loop we discussed above.
- The program control is transferred to loop update part i.e. i++ after inner loop terminates. The loops update part increment the value of i with 1 and transfer the control to loop condition.

From the above description of outer loop, it is clear that outer loop executes 10 times. For each time outer loop repeats, inner loop is executed with different value of i printing the below output.

```
1    2    3    4    5
2    4    6    8    10
3    6    9    12   15
4    8    12   16   20
5    10   15   20   25
6    12   18   24   30
7    14   21   28   35
8    16   24   32   40
9    18   27   36   45
10   20   30   40   50
```

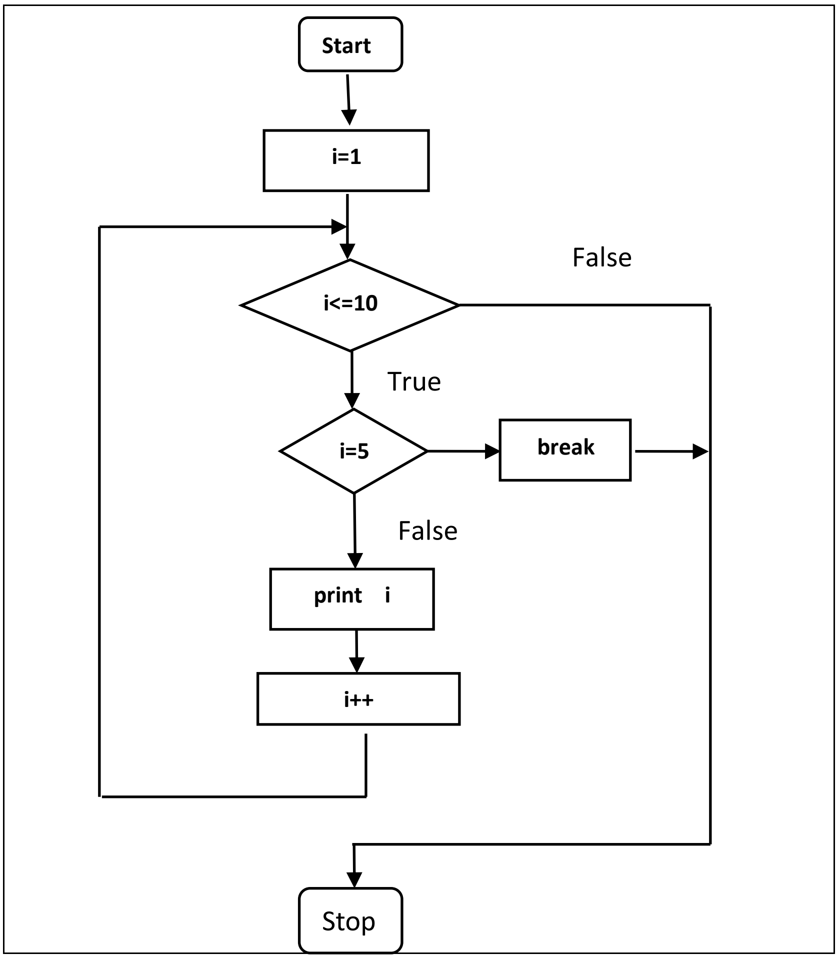
BREAK STATEMENT:

The break statement is used to terminate loops or to exit a switch. The break statement will break or terminate the inner-most loop. It can be used within a while, a do-while, a for or a switch statement. The break statement is written simply as ***break;*** without any embedded expressions or statement.
Consider this example.

EXAMPLE:

```
#include<stdio.h>
main()
{
   int i;
   for(i=1;i<=10;i++)
   {
     if(i==5)
     break;
     printf(“%d”,i);
    }
}
Output:1234
```

FLOWCHART:



CONTINUE STATEMENT :

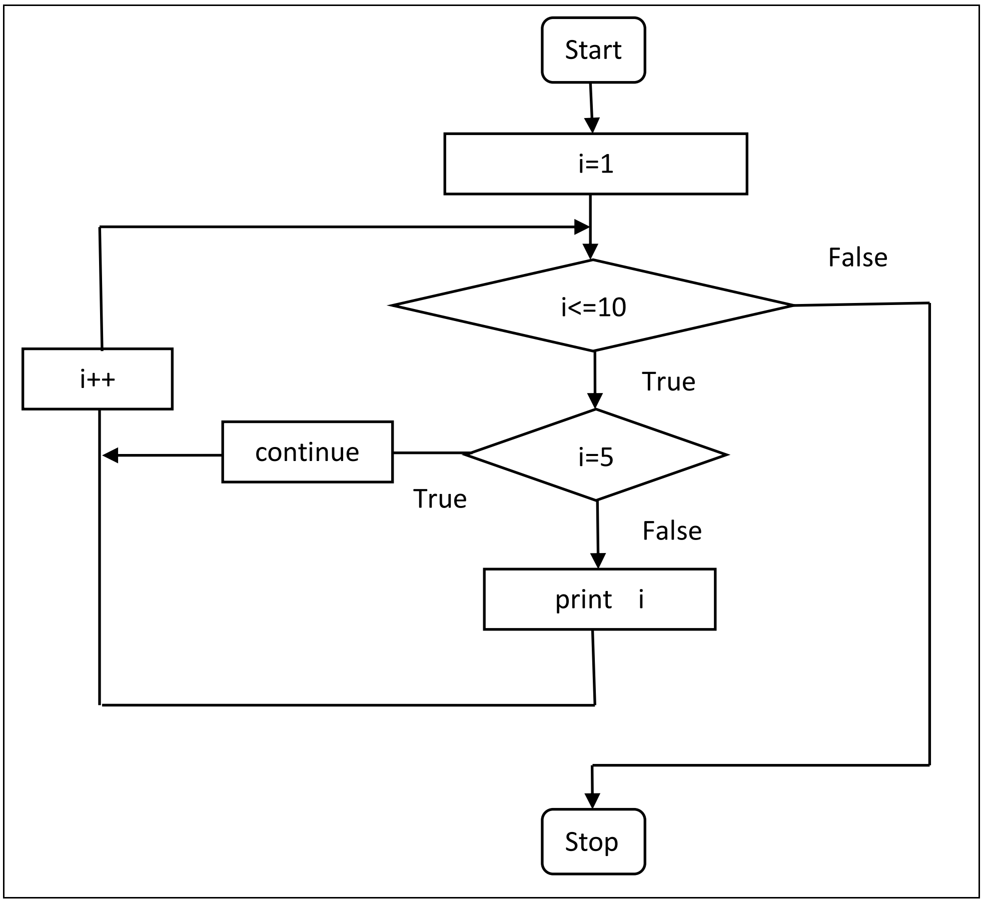
The continue statement is used to skip or to bypass some step or iteration of looping structure. It does not terminate the loop but just skip or bypass the particular sequence of the loop structure. It is simply written as ***continue;.***

EXAMPLE:

```
#include<stdio.h>
main(){
   int i;
   for(i=1;i<=10;i++)
   {
     if(i==5)
     continue;
     printf(“%d”,i);
    }
}
Output:1234678910
```

The output of the above program will be 1,2,3,4,6,7,8,9,10. The 5th iteration of the loop will be skipped as we have define the continue for that iteration. So it will not print the value '5'.

FLOWCHART:



THE GOTO STATEMENT:

The goto statement is one of C's branching, or unconditional jump, statements. Whenever the program reaches a goto statement, the execution jumps to the program's control from one statement to another statement (where label is defined). We call the goto statement is an unconditional statement because whenever the program encounters a goto statement, the execution of the program branches accordingly, which does not depend on any condition like the if statement.

Defining a label

Label is defined following by the given syntax

label_name:

- label_name should be a valid identifier name.
- **: (colon)** should be used after the label_name.

Transferring the control using 'goto'

Program's control can be transfer following by the given syntax

goto label_name;

Two styles of 'goto' statement

We can use goto statement to transfer program's control from down to top (↑) and top to down (↓).

Top to Down	Down to Top
goto label; ***----*** ***---- Forward Jump*** ***----*** ***label :*** ***statement;***	***label:*** ***----*** ***---- Backward Jump*** ***----*** ***goto label ;*** ***statement;***

Note that a goto breaks the normal sequential execution of the program. If the label: is before the statement goto label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as backward jump. On the other hand, if the label: is placed after the goto label; some statements will be skipped and the jump is known as forward jump.

EXAMPLE:

```
main()
{
          double x,y;
          read:
          scanf("%f",&x);
          if(x<0)
              goto read;
          y=sqrt(x);
          printf("%f %f\n",x,y);
    goto read;
}
```

Control may be transferred to any other statement within the program. The target statement must be labeled and the label must be followed by a colon. Thus the target statement will appear as label: statement. Each labeled statement within the program must have unique label, i.e. no two statement can have same label.

UNIT - 3

LIBRARY FUNCTION

STRING FUNCTION:

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0. (The null character has no relation except in name to the *null pointer*. In the ASCII character set, the null character is named NULL. The null or string-terminating character is represented by another character escape sequence, \0. To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions

1. strcpy( ):

This function copies the contents of one string into another. The base addresses of the source and target string should supplied to this function.

SYNTAX: ***strcpy(strDestination , strSource);***

Return Value : This function copies source string to the destination string.

Parameters: strDestination Destination string, strSource Null-terminated source string. The strcpy function copies strSource, including the terminating null character, to the location specified by strDestination.

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
void main( void )
{
  char s1[15],s2[15];
  strcpy(s1,”Computer”);
  strcpy(s2,”C Language”);
  printf(“\n%s %s”,s1,s2 );
}
```

```
Output: Computer  C Language
```

2. strncpy( ):

The strcpy function copies n characters from src to dest up to and including the terminating null character if length of src is less than n.

SYNTAX ***: char *strncpy (char *dest, char *src, int n);***

Return Value ;The strncpy function returns dest.

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char str1[]= "To be or not to be";
 char str2[6];
 strncpy (str2,str1,5);
 str2[5]='\0';
 puts (str2);
 return 0;
}
```

```
To be
```

3. strcat( ):

This function concatenates the source string at the end of the target string. For example “Bombay” and “Nagpur” on concatenation would result into a string “BombayNagpur”. Here is an example of strcat().

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char str[80];
 strcpy (str,"these ");
 strcat (str,"strings ");
 strcat (str,"are ");
 strcat (str,"concatenated.");
 puts (str);
 return 0;
}
```

```
these strings are concatenated.
```

4. strncat( ):

Syntax: ***char *strncat(char *str1, const char *str2,size_t count);***

strncat(str1,str2,n) concatenates not more than n characters of the string pointed to by str2 to the the string pointed to by str1 and terminates str1 with a null. The first character of str2 overwrites the null terminator originally ending str1. The str2 is untouched by the operation.

```
#include <stdio.h>
#include <string.h>
main(void)
 {
   char s1[10]="JJKCC",
s2[20]="BCA.BBA.BCOM.";
   clrscr();
   strncat(s2, s1, 4);
   printf(s2);
   printf("Answer is :%s",s1);
  getch();
}
```

```
Answer is : JJKCCBCA.
```

5. strchr( ):

Syntax: ***char *strchr(const char *str, int c***);

Searches for the first occurrence of the character c (an unsigned char) in the string pointed to by the argument str. The terminating null character is considered to be part of the string. Returns a pointer pointing to the first matching character, or null if no match was found.

```
#include <stdio.h>
#include <string.h>
void main()
{
  char str1[20] = "computer program";
  char * ptr;
  char   ch = 'p';
  ptr = strchr( str1, ch );
  printf( "The first occurrence of %c in '%s' is '%s'\n",ch,
str1, ptr );
  getch();
}
```

```
The first occurrence of p in computer program is computer
program
```

6. strrchr( ):

Syntax: ***char *strrchr(char *string, int c);***

The strrchr function searches string for the last occurrence of c. The null character terminating string is included in the search. The strrchr function returns a pointer to the last occurrence of character c in string or a null pointer if no matching character is found.

```
#include <stdio.h>
void main() {
  char s;
  char str1 [] = "This is a testing";
  s = strrchr (str1, 't');
  clrscr();
  if (s != NULL)
  {
    printf ("found a 't' at %s\n", s);
  }
  getch();
}
```

```
found a 't' at ting
```

7. strcmp( ):

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the string is reached, whichever occurs first. If the two string are identical, strcmp( ) returns a value zero. If they are not, it returns the numeric difference between the ASCII values of the first non – matching pairs of characters.

SYNTAX: ***int strcmp(string1, string2);***
Parameters: string1, string2 Null-terminated strings to compare

Return Value:

- if Return value if < 0 then it indicates string1 is less than string2
- if Return value if > 0 then it indicates string2 is less than string1
- if Return value if = 0 then it indicates string1 is equal to string1

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
void main( void )
{
  char s1[ ]="Jerry";
  char s2[ ]="Ferry";
  int i,j,k;
  i=strcmp(s1,"Jerry");
  j=strcmp(s1,s2);
 k=strcmp(s1,"Jerry boy");
  printf("\n%d%d%d",i,j,k);
}
```

Output: 0 4 -32

In the first call to strcmp( ),the two strings are identical – "Jerry"and "Jerry" – and the value returned by strcmp( ) is zero. In the second call, the first character of "Jerry" does not match with the first character of "Ferry" and the result is 4, which is the numeric difference between ASCII value of 'J' and ASCII value of 'F'. in the third call to strcmp( )"Jerry" does not match with "Jerry boy", because the null character at the end of "Jerry" does not match the blank in "Jerry boy". The value returned is -32 , which is the value of null character minus the ASCII value of space, i.e. '\0' minus ' ', which is equal to -32.

8. strncmp( ):

SYNTAX: ***int strncmp(const char *s1, const char *s2, int n);***
strncmp will test at most 'n' characters in s1 to s2 for equallity.

"strncmp" compares the first N characters of the string "s1" to the first N characters of the string "s2". If one or both of the strings is shorter than N characters (i.e. if "strncmp" encounters a '\0'), comparisons will stop at that point. Thus N represents the maximum number of characters to be examined, not the exact number. (Note that if N is zero, "strncmp" will always return zero -- no characters are checked, so no differences are found.)

```
#include <stdio.h>
void main()
{
        char s1[ ]="BCAPGDCA";
        char s2[ ]="BCABBA";
        clrscr();
        if(strncmp(s1,s2,3)==0)       {
                printf("3 characters are same");
        }
        else    {
                printf("3 characters are not same");
        }
        getch();
}
```

3 characters are same

9. strspn( ):

The strspn() function returns the index of the first character in string1 that doesn't match any character in string2. If all the characters of the string2 matched with the string1, strspn returns the length of the string1.

```
#include <stdio.h>
void main()
{
        char s1[ ]="BCAPGDCA";
        char s2[ ]="BCABBA";
        int n;
        clrscr();
        n=strspn(s1,s2);
        printf("The number length is
:%d",n);
        getch();
}
```

The number length is :4

10.strcspn( ):

SYNTAX: size_t strcspn(const char *str1, const char *str2);

strcspn is the function from c standard library (**header file string.h**).

It search the string for certain set of characters. The **strcspn()** function calculates the length of initial segment of string 1 which does not contain any character from string 2. This function returns the index of first character in string 1 that matches with any character of string2.

```
#include <stdio.h>
#include <string.h>
void main()
{
        char s[20] = "JJKCC007", t[20] = "0123456789";
        int n;
        n=strcspn(s, t);
        printf("The first decimal digit of s is at position: %d.\n",n );
        getch();
}
```

```
The first decimal digit of s is at position:5
```

11.strlen( ):

This function counts the number of characters present in a string.

SYNTAX: ***strlen(string);***

Return Value: This function returns the number of characters in string, excluding the terminal NULL.

Parameter:string Null - terminated string.

EXAMPLE:

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
void main( void )
{
  char buffer[61] = "How long am I?";
  int  len;
  len = strlen( buffer );
  printf( "'%s' is %d characters long\n", buffer, len );
}
```

```
Output: 'How long am I?' is 14 characters long
```

12.strpbrk( ):

SYNTAX: ***char strpbrk (const char *s, const char *charset)***

The **strpbrk** function locates in the null-terminated string s the first occurrence of any character in the string charset and returns a pointer to this character. If no characters from charset occur anywhere in s **strpbrk** returns NULL.

```
#include <stdio.h>
#include <string.h>
void main()
{
        char s[20] = "JJKCC1010101", t[3] = "01";
        int no;
        clrscr();
        no=(strpbrk(s,t)-s);
        printf("The position of the first binary digit of s is: %d.\n",no);
        getch();
}
```

```
The position of the first binary digit of s is:5
```

13.strstr( ):

SYNTAX: ***char *strstr(char *string2, char string*1);***

The strstr function locates the first occurrence of the string string1 in the string string2 and returns a pointer to the beginning of the first occurrence.The strstr function returns a pointer within string2 that point to a string identical to string1. If no such sub string exists in src a null pointer is returned.

```
#include <stdio.h>
void  main()
{
  char s1 [] = "My House is small";
  char s2 [] = "My Car is green";
  clrscr();
  printf ("Returned String 1: %s\n", strstr (s1, "House"));
  printf ("Returned String 2: %s\n", strstr (s2, "Car"));
  getch();
}
```

```
Returned String 1: House is small
Returned String 2: Car is green
```

14.strtok( ):

SYNTAX: ***char *strtok(char *str1, const char *str2);***

The strtok() function returns a pointer to the next "token" in str1, where str2 contains the delimiters that determine the token. strtok() returns NULL if no token is found. In order to convert a string to tokens, the first call to strtok() should have str1 point to the string to be tokenized. All calls after this should have str1 be NULL.

```
#include <stdio.h>
#include <string.h>
void  main()
{
 char string[] = "KUNDALIA #COMMERCE  #COLLEGE";
 char* token;
 char dlm[] = "#";
 token = strtok(string,dlm);
 while( token != NULL)
 {
   printf("%s\n", token);
   token = strtok(NULL,"#");
 }
 getch();
}
```

```
KUNDALIA
COMMERCE
COLLEGE
```

MATHEMATICAL FUNCTIONS:

1. **acos( ):**

SYNTAX: ***double acos (double x)***

Acos is used to find the arccosine of a number (give it a cosine value and it will return the angle, in radians corresponding to that value). It must be passed an argument between -1 and 1.

```
#include <stdio.h>
#include <math.h>
int main()
{
  double x = 0.5;
  double result = acos(x);
  printf("The arc cosine of %lf is %lf\n",
x, result);
  return 0;
}
```

```
The arc cosine of 0.500000 is 1.047198
```

2. **asin( ):**

SYNTAX: ***double asin (double x);***

The value of x must be within the range of -1 to +1 (inclusive).

Return value: The returned value is in the range of -pi/2 to +pi/2 (inclusive).

```
#include <stdio.h>
#include <math.h>
int main(void)
{
  double result, x = 0.289;
  clrscr();
  result = asin(x);
  printf("The arc sin of %lf is %lf\n", x,
result);
  getch();
 }
```

```
The arc sin of 0.289000 is 0.293182
```

3. **atan( ):**

SYNTAX: ***double atan(double arg);***

Calculates the arctangent of *x* (**atan** or **atanf**) or the arctangent of *y*/*x* (**atan2** or **atan2f**).

```
#include <stdio.h>
#include <math.h>
void main()
{
  double result, x = 0.6325;
  clrscr();
  result = atan(x);
  printf("The arc sin of %lf is %lf\n", x, result);
  getch();
}
```

```
The arc sin of 0.632500 is 0.563974
```

4. **ceil( ):**

returns the smallest integer not less than num.

SYNTAX: ***double ceil(double x);***

Return Value The **ceil** function returns a **double** value representing the smallest integer that is greater than or equal to *x*. There is no error return.

Parameter **x** Floating-point value

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    printf("%f", ceil(9.9));
}
```

```
output : 10.000000
```

5. cos( ):

double cos (double x);

The **cos** function computes the cosine of *x* (measured in radians). It returns a value in the range of -1 to 1.

```
#include <stdio.h>
#include <math.h>
int main()
{
  double x = 1.047198;
  double result = cos(x);
  clrscr();
  printf("The cosine of %lf is %lf\n", x, result);
   getch();
}
```

```
The cosine of 1.047198 is 0.500000
```

6. div( ):

SYNTAX: div_t div (int n, int d);

This function of stdlib.h will return the quotient and remainder of the division of numerator by denominator integers n,d.

```
#include<stdio.h>
#include<stdlib.h>
void  main ()
{
        div_t div_res;
        div_res = div (18,7);
        clrscr();
        printf("18   div   7   =%d,remainder   %d",div_res.quot,
div_res.rem);
        getch();
}
```

```
18 div 7 = 2 remainder 4
```

7. exp( ):

Calculates the exponential.

SYNTAX : double exp(double x);

Return Value : The exp function returns the exponential value of the floating-point parameter, x, if successful. On overflow, the function returns INF (infinite) and on underflow, exp returns 0.

Parameter :x Floating-point value

```
#include <math.h>
#include <stdio.h>
void main( void )
{
  double x = 2.302585093, y;
  y = exp( x );
  printf( "exp( %f ) = %f\n", x, y );
}
```

```
Output: exp( 2.302585 ) = 10.000000
```

8. fabs( ):

Calculates the absolute value of the floating-point argument.

SYNTAX : double fabs(double x);

Return Value :fabs returns the absolute value of its argument.

Parameter x Floating-point value

```
#include<stdio.h>
#include<math.h>
main(){
  float a=12.44,b-12.44;
  printf(“\n a=   %f”,fabs(a));
  printf(“\n b=   %f”,fabs(b));
}
```

```
Output is:
a=12.44
b=12.44
```

9. floor( ):

Calculates the floor of a value.

SYNTAX: ***double floor(double x);***

Return Value :The floor function returns a floating-point value representing the largest integer that is less than or equal to x. There is no error return.

Parameter: x Floating-point value

```
#include <math.h>
#include <stdio.h>
 int main(void)
 {
   printf("%f", floor(10.9));
 }
OUTPUT: 10.000000
```

10.fmod( ):

Calculates the floating-point remainder.

SYNTAX: ***double fmod(double x, double y);***

Return Value :fmod returns the floating-point remainder of x / y.

Parameters:x, y Floating-point values

```
#include <math.h>
#include <stdio.h>
 void main(void)
  {
    printf("%1.1f", fmod(10.0, 3.0));
   }
Output : 1.0
```

11.log( ):

Returns the natural logarithm of a number.

SYNTAX: ***log(number);***

The number argument can be any valid numeric expression greater than 0.

```
#include <stdio.h>
#include <math.h>
int main ()
{
  double param, result;
  param = 5.5;
  result = log (param);
  printf ("ln(%lf) = %lf\n", param, result );
  return 0;
}
ln(5.500000) = 1.704748
```

12.modf( ):

***SYNTAX:* double modf (double value, double *iptr);**

The **modf** functions break *value* into the integral and fractional parts, each of which has the same sign as the argument. They return the fractional part, and store the integral part (as a floating-point number) in the object pointed to by *iptr*.

```
#include <math.h>
#include <stdio.h>
Void main() {
  double x = 31415.9265, fractional, integer;
  fractional = modf(x, &integer);
  clrscr();
  printf("%.4lf = %.4lf + %.4lf\n", x, integer, fractional);
  getch();
}
31415.9265 = 31415.0000 + 0.9265
```

13.pow( ):

Calculates x raised to the power of y.

SYNTAX: **double pow(double *x*, double *y*);**

Return Value :pow returns the value of xy. No error message is printed on overflow or underflow.

Parameters :x Base, y Exponent

EXAMPLE:

```
#include <math.h>
#include <stdio.h>
void main( void ) {
  double x = 2.0, y = 3.0, z;
  z = pow( x, y );
  printf( "%.1f to the power of %.1f is %.1f\n", x, y, z );
}
```

```
Output : 2.0 to the power of 3.0 is 8.0
```

14.sin( ):

double sin (double x)

The **sin** functions compute the sine of *x* (measured in radians).

```
#include <stdio.h>
#include <math.h>
int main(void) {
 double x = 0.31415926;
 double result = sin(x);
 printf("The sine of %lf is %lf\n", x, result);
 return 0;
}
```

```
The sine of 0.314159 is 0.309017
```

15.sqrt( ):

Calculates the square root.

SYNTAX : *sqrt(n);*

Return Value :The sqrt function returns the square-root of n. If x is negative, sqrt returns an indefinite.

Parameter n Nonnegative numeric value

```
#include <math.h>
#include <stdio.h>
void main( void ) {
  float y;
  int x = 81;
  y = sqrt( x );
  printf( "sqrt( %d ) = %f\n", x, y );
}
```

```
Output: exp( 81 ) = 9.000000
```

DATE & TIME FUNCTIONS:

1. clock( )

SYNTAX : *clock_t clock (void);*

The **clock** function determines the amount of processor time used since the invocation of the calling process, measured in *CLOCKS_PER_SEC* of a second.

The code below counts the number of seconds that passed while running some for loop.

```
#include <stdio.h>
#include <time.h>
#include <math.h>
int main()
{
 clock_t start = clock();
 long i;
 for (i = 0; i < 100000000; ++i)
 {
  exp(log((double)i));
 }
 clock_t finish = clock();
 printf("It took %d seconds to execute the for loop.\n",
             (finish - start) / CLOCKS_PER_SEC);
 return 0;
}
```

```
It took 23 seconds to execute the for loop.
```

2. difftime( )

SYNTAX : ***double difftime (time_t time2, time_t time1);***

Time1 and time2 are both time_t objects.

Return Value: The difference in seconds between time2-time1 will be returned as a floating point double.

```
#include <stdio.h>
#include <time.h>
void  main()
{
  time_t start, end;
  char str[256];
  double dif;
  clrscr();
  time (&start);
  printf ("Please, enter college name: ");
  gets (str);
  time (&end);
  dif = difftime (end, start);
  printf ("You have taken %.2lf seconds to type college name.\n", dif );
  getch();
}
```

```
Please, enter college name:jj kundalia commerce college
You have taken 10 seconds to type college name.
```

3. mktime( )

SYNTAX : ***time_t mktime (timeptr);***

The **mktime** function converts the local time into a calendar value. The ***timeptr*** argument points to a structure that contains the local time. The structure is described in the reference page for **asctime**. The converted time has the same encoding as the values returned by the ***time*** function.

The **mktime** function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value -1 (**time_t**).

```
#include <time.h>
 #include <stdio.h>
 int main(void)
 {
   struct tm t;
   time_t t_of_day;
   t.tm_year = 2005-1900;
   t.tm_mon = 0;
   t.tm_mday = 3;
   t.tm_hour = 0;  /* hour, min, sec don't matter */
   t.tm_min = 0;   /* as long as they don't cause a */
   t.tm_sec = 1;   /* new day to occur */
   t.tm_isdst = 0;
   t_of_day = mktime(&t);
   printf(ctime(&t_of_day));
   return 0;
 }
```

```
Mon Jan  3 00:00:01 2005
```

4. time( )

SYNTAX : ***time_t time(time_t *time);***

The function time() returns the current time, or -1 if there is an error. If the argument time is given, then the current time is stored in time.

```
#include <time.h>
#include <stdio.h>
void main(void) {
   struct tm *ptr;
   time_t lt;
   clrscr();
   lt = time(NULL);
   ptr = localtime(&lt);
   printf(asctime(ptr));
   getch();
}
```

```
Sat Mar  3 16:12:48 2007
```

5. asctime( )

SYNTAX : ***char *asctime(const struct tm *timeptr);***

The *asctime*() function shall convert the broken-down time in the structure pointed to by *timeptr* into a string in the form:

Day month date hours : minutes : seconds year
Sun Sep 16 01:03:52 1973

```
#include <time.h>
#include <stdio.h>
int main(void) {
   struct tm *ptr;
   time_t lt;
   lt = time(NULL);
   ptr = localtime(&lt);
   printf(asctime(ptr));
   return 0;
}
```

```
Sat   Mar  3 16:12:43 2007
```

6. ctime( )

Converts the time_t object pointed by *timer* to a C string containing a human-readable version of the corresponding local time and date.

The returned string has the following format: Www Mmm dd hh:mm:ss yyyy

Where Www is the weekday, Mmm the month in letters, dd the day of the month, hh:mm:ss the time, and yyyy the year. The string is followed by a new-line character ('\n') and the terminating null-character.

This function is equivalent to: asctime(localtime(timer)).

```
#include <time.h>
#include <stdio.h>
int main(void)
 {
  time_t lt;
  lt = time(NULL);
  printf(ctime(&lt));
  return 0;
 }
```

```
Sat Mar  3 16:12:45 2007
```

7. gmtime( )

SYNTAX : ***struct tm *gmtime(const time_t *time);***

Uses the value pointed by *timer* to fill a tm structure with the values that represent the corresponding time, expressed as UTC (or GMT timezone).

Returns time in Coordinated Universal Time (UTC)(essentially Greenwich mean time).

```
#include <time.h>
#include <stdio.h>
 int main(void)
 {
  struct tm *local, *gm;
  time_t t;
  t = time(NULL);
  local = localtime(&t);
  printf("Local time and date: %s\n", asctime(local));
  gm = gmtime(&t);
  printf("Coordinated Universal Time and date: %s", asctime(gm));
  return 0;
 }
```

```
Local time and date: Sat Mar  3 16:12:46 2007
Coordinated Universal Time and date: Sat Mar  3 16:12:46 2007
```

8. localtime( )

SYNTAX : ***struct tm * localtime (const time_t * ptr_time);***

The pointer ptr_time is a pointer to a time_t object that contains a calendar time.The function localtime() returns a pointer to a tm structure. The tm structure contains the time information.

The function localtime() uses the time pointed by the pointer ptr_time to fill a tm structure with the values that represent the corresponding local time.

```
#include <time.h>
#include <stdio.h>
void main(void)
{
   struct tm *local;
   time_t t;
   clrscr();
   t = time(NULL);
   local = localtime(&t);
   printf("Local time and date: %s\n", asctime(local));
   getch();
}
```

```
Local time and date: 03 10:39:00 2012
```

9. strftime( )

SYNTAX : ***strftime(string,maxlen,format,timestruct);***

The function strftime() formats date and time information from time to a format specified by fmt, then stores the result in str (up to maxsize characters). Certain codes may be used in fmt to specify different types of time

The strftime() function returns the number of characters put into str, or zero if an error occurs.

```
#include <time.h>
#include <stdio.h>
void main(void)
{
   struct tm *ptr;
   time_t lt;
   char str[80];
   clrscr();
   lt = time(NULL);
   ptr = localtime(&lt);
   strftime(str, 100, "It is now %H %p.", ptr);
   printf(str);
   return 0;
}
```

```
It is now 16 PM.
```

I/O FORMATTING FUNCTIONS:

IMPORTANCE OF HEADER FILES :

The #include directive copies the contents of an external file at a given location of the source program almost all c program use library function and require header files to be included. “stdio.h”, “conio.h”,”stdlib.h” etc are header file.

These filese are available in a separate subdirectory. All the compilers have on installation program during the process of installation. The information is supplied about the subdirectory in which the headers file available. When we want the header file”stdio.h” to be applied at the top of our program we starts the program with following line.

#include<stdio.h>

Here note that the name of header file is written between the character < >.Thus the compiler search the file “stdio.h” in the subdirectory where other header files are already located. This information has already been supplied doing the installation of a c complier.

INTRODUCTION TO SOME POPULAR HEADER FILES AND ITS LIBRARY FUNCTIONS:

- **Stdio.h:**

When we write a program, either in the declaration or any other of program, we can assign a value to a variable through an assignment statement. The value assigned this way for the first time to any variable is called the initialization. Similarly to assign the values to the variable during the execution of the program is called INPUT operation. This input can be possible through any of the input devices like key board ,file etc. the key board is considered as a standard input device and therefore any input will be from keyboard, if an input device is not specified. C provides several functions, stored in a C library, for this purpose. These functions are commonly called standard I/O functions. Hence it is required to include this library in the program, when we use these functions in the program. We do using the following statement.

#include<stdio.h>

The name stdio.h stands for standard input – output header file.

1. prinft( ):

printf() function is used to display something on the console or to display the value of some variable on the console The general syntax for printf() function is as follows.

printf(<”format string”>,<list of variables>);

To print some message on the screen

printf(“God is great”);

This will print message “God is great” on the screen or console.

Example:
printf(“\nIndia is the best”);
printf(“\nVande Matram”);
printf(“\nJay Hind”);
	Output:
	India is the best
	Vande Matram
	Jay Hind

To print the value of some variable on the screen

Integer Variable :

int a=10;
printf(“%d”,a);

Here %d is format string to print some integer value and a is the integer variable whose value will be printed by printf() function. This will print value of a “10” on the screen. You can make this output interactive by writing them

Int a=10;
printf(“a=%d”,a);

This will print "a=10" on the screen

To print multiple variable's value one can use printf() function in following way.

Int p=1000,r=10,n=5;
printf("amount=%d rate=%d yeat=%d",p,r,n);

This will print "amount=1000 rate=10 year=5" on the screen

float Variable :
float per=70.20;
printf("Percentage=%f",per);

Here %f is format string to print some float(real) value and per is the float variable whose value will be printed by printf() function. This will print value of a "Percentage=70.20" on the screen. Character Variable :

char ans='Y';
printf("Answer=%c",ans);

Here %c is format string to print single character value and ans is the character variable whose value will be printed by printf() function. This will print value of a "Answer=Y" on the screen.

Suppose we want to print "Amar" on the screen with character variable

Char c1='A',c2='m',c3='a',c4='r' ;
Printf("Name = %c %c %c %c",c1,c2,c3,c4);

This will print "Name=A m a r" on the screen.

You can use single printf() function to print different data type variable's value also.

EXAMPLE :

int rno=10;
char res='P';
float per=75.70;
printf("Rollno=%d Result=%c Percentage=%f",rno,res,per);

This will print message "Rollno=10 Result=P Percentage=75.70" on the screen.

2. Scanf( ) :

Scanf() function is use to read data from keyboard and to store that data in the variables.

The general syntax for scanf() function is as follows.

Scanf("Format String",&variable);

Here format string is used to define which type of data it is taking as input this format string can be %c for character, %d for integer variable and %f for float variable. Whereas variable the name of memory location or name of the variable and & sign is an operator that tells the compiler the address of the variable where we want to store the value. One can take multiple input of variable with single scanf() function but it is recommended that there should be one variable input with one scanf() function.

Scanf("Format string1,format string2",&variable1,&variable2);

For Integer Variable :

int rollno;
scanf("%d",&rollno);
printf("Enter rollno=");

Here in scanf() function %d is a format string for integer variable and &rollno will give the address of variable rollno to store the value at variable rollno location..

For Float Variable :

float per;
printf("enter percentage=");
scanf("%f",&per);

Here in scanf() function %f is a format string for float variable and &per will give the address of variable per to store the value at variable per location.

For character variable:

char ans;
printf("enter answer=");
scanf("%c",&ans);

Here in scanf() function %c is a format string for character variable and &ans will give the address of variable ans to store the value at variable ans location.

3. getc():

int getc(FILE *stream);

The *getc*() function shall be equivalent to *fgetc*() , except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

```
#include<stdio.h>
#include<conio.h>
void main(){
            int i;
            char ans;
            clrscr();
            printf("Enter a character:");
            ans=getc(stdin);
            printf("%c",ans);
            getch();
}
```

```
OUTPUT
Enter a character: A
A
```

4. getchar( ):

getchar() function will accept a character from the console or from a file, displays immediately while typing and we need to press Enter key for proceeding.

Syntax : **int getchar(void);**

It returns the unsigned char that they read.If end-of-file or an error is encountered getchar() functions return EOF.

example

char a;
a= getchar();

The function "getchar()" reads a single character from the standard input device, the keyboard being assumed because that is the standard input device, and assigns it to the variable "a".

```
#include <stdio.h>
 int main()
{
   char ans;
   puts("Type your favorite keyboard character:");
   ans=getchar();
   printf("Your favorite character is %c\n", ans);
  return(0);
}
```

```
Type your favorite keyboard character: A
    Your favorite character is A
```

5. gets( ) :

We call a group of character as a string. A string in C is enclosed within double quotation. It always ends with a null character ('\0'). We can use a sequence of getchar()

function to accept a string. Alternatively we use gets( ) function to read a string. The syntax of gets( ) function is

gets(variable_name);

When this statement is encountered by the C compiler, it will wait for the user to enter sequence of character from the input device (if not specified, it is key board). The wait gets automatically terminated when an enter key is pressed. A null character is automatically assigned as a last character of the string by the compiler, immediately after the enter key is pressed. After execution of this statement, the variable_name is used to access the content of string.

```
#include<stdio.h>
void main()
{
  char str1[80];
  printf("please type a string....\n");
  gets(str1);
  printf("string is %s\n",str1);
}
```

```
output:
please type a string….
This is my First string.
string is: This is my First string
```

In the above example the keyed –in string will be stored in the variable str1. The printf statement will display the content of the string variable str1.

6. putc():

int putc (int character, FILE * stream);

Writes a character to the stream and advances the position indicator. The character is written at the current position of the stream as indicated by the internal position indicator, which is then advanced one character.

```
#include <stdio.h>
int main ()
{
  FILE * pFile;
  char c;
  pFile=fopen("alphabet.txt","wt");
  for (c = 'A' ; c <= 'Z' ; c++)
 {
   putc (c , pFile);
 }
 fclose (pFile);
 return 0;
}
```

7. Putchar():

int putchar (int character);

Writes *character* to the current position in the standard output (stdout) and advances the internal file position indicator to the next position. If there are no errors, the same character that has been written is returned.

If an error occurs, EOF is returned and the error indicator is set.

```
#include <stdio.h>
int main ()
{
  char c;
  for (c = 'A' ; c <= 'Z' ; c++)
    {
      putchar (c);
    }
  return 0;
}
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

8. puts( )

We use puts( ) function to write a string to an output device. The syntax of puts ( ) function is

puts(variable name);

puts( ) function writes the content of the string stored in a variable name to the monitor till the null character is encountered. Recall the gets( ) function, where a null character is automatically stored as a last character of the string. Note that every string contains a null character but puts( ) will not display this character.

9. fflush( )

It is designed to remove or 'flush out' any data remaining in the buffer. The argument to fflush( ) must be the buffer which we want to flush out. Here we use 'stdin', which means buffer related with standard input device – keyboard.

10.getch( ):

The function getch( ) is used for inputting a character but the difference is that the character keyed in will not be echoed on the screen. i.e. the character is not visible on the screen. If the end-user does not want to show the character keyed in, this function is used.

character variable=getch();

11.getche( ):

This function return the character that has been most recently typed.The function getche( ) is used for inputting a character but the difference is that the character keyed in will be echoed on the screen. The 'e' in getche( ) function means it echoing it on screen.

character variable=getche();

12.gotoxy( ):

This function place the cursor at appropriate position on the screen. The parameter passed to gotoxy( ) are column number followed by row number.

gotoxy(c,r);

Suppose we write.

gotoxy(30,12);
printf("Computer Center");

These statements will print computer center at column number 30 and row number 12.

13.ungetc ( ):

int ungetc(int c, FILE *stream);

The **ungetc** function pushes the character c (converted to an unsigned char) back onto the input stream pointed to by stream. The pushed-back characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions (fseek) will discard the pushed back characters.

MISCELLANEOUS FUNCTIONS:

1. delay( ):

void delay(unsigned int);

delay function is used to suspend execution of a program for a particular time.

Here unsigned int is the number of milliseconds (remember 1 second = 1000 milliseconds). To use delay function in your program you should include the dos.h header file.

```
#include<stdio.h>
#include<stdlib.h>
 main()
{
   printf("This c program will exit in 10
seconds.\n");
   delay(10000);
   return 0;
}
```

Above c program exits in ten seconds,

after the printf function is executed the program waits for 10000 milliseconds or 10 seconds and then program termination occurs.

2. clrscr( ):

This function is useful to clear the screen. It is stored in the "conio.h" header file.

<u>SYNTAX:</u> **clrscr();**

clrscr(); :- This is used for clearing the output screen i.e console

suppose you run a program, alter it and run it again you may find that the previous output is still stuck there itself, at this time clrscr(); would clean the previous screen. One more thing to remember always use clrscr(); after the declaration like

```
int a,b,c;
float total;
clrscr();
```

3. clear( ):

Clears the value of a single variable. Also clears all the filters that were set if the variable is a record and resets the key to the primary key.

CLEAR(Variable)

Variable : Any

The identifier (variable) of any C/AL data type, including simple and composite data types. The following rules apply when you run the CLEAR function:

- A number variable is set to 0 (zero)
- A string variable is set to empty string
- A date variable is set to 0D (undefined date)
- A time variable is set to 0T (undefined time)
- A Boolean variable is set to FALSE

4. errno( ):

errno.h is a header file in the standard library of C programming language. It defines macros to report error conditions through error codes stored in a static location called errno.

A value is stored in errno by certain library functions when they detect errors. At program startup, the value stored is zero. Library functions store only values greater than zero. Any library function can alter the value stored before return, whether or not they detect errors. Most functions indicate that they detected an error by returning a special value, typically NULL for functions that return pointers, and -1 for functions that return integers. A few functions require the caller to preset errno to zero and test it afterwards to see if an error was detected.

5. isalnum( )

<u>SYNTAX</u> : *int isalnum(int ch)*

<u>Return:</u> returns nonzero if its argument is either a letter of the alphabet or a digit. Ret urns zero if the character is not alphanumeric.

```
#include <ctype.h>
#include <stdio.h>
int main(void) {
        char ch;
        ch = getc(stdin);
     if(isalnum(ch)
        printf("%c is alphanumeric\n", ch);
}
```

Output : 1 is alphanumeric

6. isalpha( )

SYNTAX : *int isalpha(int ch)*

Return: returns nonzero if ch is a letter or zero if not.

EXAMPLE:

```
#include <ctype.h>
#include <stdio.h>
void  main(void)  {
  char ch;
  ch = getchar();
 if(isalpha(ch))
          printf("%c is a letter\n", ch);
   }
```

```
Output :d
d is a letter
```

7. iscntrl( )

int iscntrl(int ch);

The iscntrl() function returns non-zero if its argument is a control character (between 0 and 0x1F or equal to 0x7F). Otherwise, zero is returned.

```
#include <stdio.h>
int main()
 {
  if( iscntrl( 'a' ) )  {
    printf( "Character a is not control character\n" );
  }
  if( iscntrl( '\n' ) )  {
    printf( "Character \n is control chracter\n" );
  }
  return 0;
}
```

```
Character \n is control chracter
```

8. isdigit( )

SYNTAX *:int isdigit(int ch)*

Return: returns nonzero if ch is a digit or zero is returned.

EXAMPLE:

```
#include <ctype.h>
 #include <stdio.h>
void  main(void)
 {
     char ch;
     ch = getchar();
            if(isdigit(ch))
     printf("%c is a digit\n", ch);
 }
```

```
Output: 3 is a digit
```

9. isgraph( )

int isgraph(int *c*);

This function is used to check given character has a graphical representation or not. It determines the characters with graphical representation which can be printed or not. Whitespace characters are not considered as a isgraph characters. It returns zero if character has a graphical representation otherwise returns nonzero.

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

```
            int i;
            char ch;
            clrscr();
            printf("\n Enter any character : ");
            ch = getchar();
            if(isgraph(ch))
                        printf("\n %c is eligible for printing !", ch);
            else
                        printf("\n %c is not eligible for printing !", ch);
            getch();
            return 0;
}
```

```
Enter any character :a
a is eligible for printing !"
```

10. islower( )

SYNTAX: ***int islower(int ch)***

Return: returns nonzero if ch is a lowercase letter; otherwise zero is returned.

```
#include <ctype.h>
#include <stdio.h>
void  main(void)
{
        char ch;
        ch = getchar();
        if(islower(ch)
        printf("%c is lowercase\n", ch);
 }
```

```
Output: b is lowercase
```

11. isprint( )

SYNTAX :***int isprint(int ch)***

Return: returns nonzero if ch is a printable character, including a space; otherwise zero is returned.

```
#include <ctype.h>
 #include <stdio.h>
 void main(void)
 {
            char ch;
            ch = getchar();
       if(isprint(ch))
      printf("%c is printable\n",ch);    }
```

```
output: a is printable
```

12.isspace( )

SYNTAX : ***int isspace(int ch)***

Return: returns nonzero if ch is a white space character, including space, horizontal tab, vertical tab, form feed, carriage return, or newline character; otherwise zero is returned.

```
#include <ctype.h>
 #include <stdio.h>
 void main(void)
 {    char ch;
          ch = getchar();
    if(isspace(ch))
          printf("%c is white space\n", ch);
  }
```

```
output :   is white space
```

13.isupper()

SYNTAX :int isupper(int ch)

Return: returns nonzero if ch is an uppercase letter; otherwise zero is returned.

EXAMPLE:

```
void main(void)
 {
            char ch;
            ch = getchar();
           if(isupper(ch))
                     printf("%c is uppercase\n", ch);
}
```

```
OUTPUT:  C
C is uppercase
```

14.isxdigit()

This function is used to check given digit is a hexadecimal digit or not. It can be any from A-F, a-f, 0-9.

```
#include <stdio.h>
#include <ctype.h>
int main ()
{
   int i;
   char ch;
   clrscr();
   printf("n Enter any hexamdecimal digit : ");
   ch = getchar();
   if isxdigit(ch)
      printf("n It is hexadecimal digit !");
   else
      printf("n It is not hexadecimal digit !");
   getch();
   return 0;
}
```

```
Enter any hexamdecimal digit :c
It is hexadecimal digit !
```

15.toupper( );

SYNTAX :int toupper(int ch)

Return: returns the uppercase equivalent of ch if ch is a letter or ch is returned unchanged.

```
void  main(void)
{
 putchar(toupper('a'));
}
```

```
OUTPUT: A
```

16.tolower( )

17. **SYNTAX : *int tolower(int ch)***

Return: returns the lowercase equivalent of ch if ch is a letter or ch is returned unchanged.

```
void  main(void)
{
   putchar(tolower('Q'));
}
```

```
OUTPUT: q
```

STANDARD LIBRARY FUNCTIONS:

1. abs( ):

Returns the absolute, positive value of the given numeric expression.

SYNTAX ***abs(numeric_expression)***

Arguments numeric expression: Is an expression of the integer data type category.

Return Types: The abs function returns the absolute value of its parameter

EXAMPLE:

```
#include<stdio.h>
#include<math.h>
void main()
{
  int a=12,b-12;
  printf(“\n a=   %d”,abs(a));
  printf(“\n b=   %d”,abs(b));
}
```

```
a=12
b=12
```

2. atof():

double atof (const char * str);

C string str interpreting its content as a floating point number. White-spaces are discarded until the first non-whitespace character is found. A valid floating point number for atof is formed by: plus or minus sign, sequence of digits, decimal point and optional exponent part (character ‘e’ or ‘E’).

The function returns the converted floating point number as a double value, if successful. If no valid conversion could be performed then an zero value is returned.

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{
  char x[10] = "3.141592";
  double pi = atof(x);
  printf("pi = %lf\n", pi);
  return 0;
}
```

```
pi = 3.141592
```

3. atol():

long atol (const char *nptr);

The **atol** function converts the initial portion of the string pointed to by *nptr* to *long* integer representation. It is equivalent to

The function returns the converted integral number as an int value, if successful and returns it as a long int. If no valid conversion could be performed then an zero value is returned. If the value is out of range then LONG_MAX or LONG_MIN is returned.

```
#include <stdio.h>
#include <stdlib.h>
 int main() {
  char x[20] = "2147111222";
  long max = atol(x);
  printf("max = %ld\n", max);
  return 0;
}
```

```
max = 2147111222
```

4. exit():

void exit (int status);

On call the process will terminate normally. It will perform regular cleanup as normal for a normal ending process. The argument status is returned to the host environment. Normally you say 1 or higher if something went wrong and 0 if everything went ok. For example exit(o) or exit (1).

5. free():

void free (void * ptr);

If you have allocated a memory block with the functions malloc(), calloc() or realloc() then you need to free the previously allocated memory. The parameter is a pointer to a memory block that was allocated with malloc(), calloc() or realloc(). If you pass a null pointer then there will no action occur. No return value by the function free().

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

```
{
        int * buffer;
        /*get a initial memory block*/
        buffer = (int*) malloc (10*sizeof(int));
        /*free initial memory block*/
        free (buffer);
        return 0;
}
```

6. labs():

long int labs (long int n);

This function will return the absolute value of a long integer.

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{
  long int a = -231564565;
  printf("|a| = %ld\n", labs(a));
  return 0;
}
```

```
|a| = 231564565
```

7. qsort():

*void qsort (void * base, size_t num, size_t size, int (* comparator) (const void *, const void *));*

The function qsort() of the stdlib.h is used to sort the elements of an array.

The function qsort() sorts the num element of the array pointed by base. Every element has a size of size bytes long. The qsort function will use the comparator function to determine the order of the elements.

The function will not return a value. The content is modified in the newly sorted order. The base parameter is a pointer to the first element of the array. The num parameter is the number of elements in the array. The size parameter gives the size in bytes of each element in the array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ()
{
  char strings[4][20] = {"apples", "grapes",
"strawberries", "bananas"};
   // sort the strings
  qsort(strings, 4, 20, (int(*)(const void*,
const void*))strcmp);
   // display the strings in ascending
lexicographic order
  for (int i = 0; i < 4; ++i)
    printf("%s\n", strings[i]);
   return 0;
}
```

```
OUTPUT

apples
bananas
grapes
strawberries
```

8. rand():

int rand (void);

The function rand() returns a pseudo-random integral number.

This number will be in the range 0 to RAND_MAX. The algorithm of rand() uses a seed to generate the series of numbers, this is why srand must be used to initialize the seed to some distinctive value. The constant RAND_MAX is defined in standard library (stdlib). The random numbers are delivered from a predetermined range. Will return an integer value between 0 and RAND_MAX.

9. strtoul():

unsigned long int strtoul (const char * str, char ** endptr, int base);

C string str interpreting its content as an unsigned integral number of the specified base, which is returned as an unsigned long int value. The function returns the converted floating point number as a unsigned long int value, if successful. If no valid conversion could be performed then an zero value is returned.

```
#include<stdio.h>
#include<stdlib.h>
int main ()
{
		char buffer[256];
		unsigned long ul;
		printf ("Enter an unsigned number: ");
		fgets (buffer,256,stdin);
		ul = strtoul (buffer,NULL,0);
		printf ("Output: %lu\n",ul);
		return 0;
}
```

```
Enter an unsigned number: 12
Output: 12
```

10.srand():

void srand (unsigned int seed);

The function srand() is used to initialize the pseudo-random number generator by passing the argument seed.Often the function time is used as input for the seed. If the seed is set to 1 then the generator is reinitialized to its initial value. Then it will produce the results as before any call to rand and srand.

```
 #include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main ()
{
		printf ("Our first number: %d\n", rand() % 100);
		srand ( time(NULL) );
		printf ("Some random number: %d\n", rand() % 100);
		srand ( 1 );
		printf ("The first number again: %d\n", rand() %100);
		return 0;
}
```

```
OUTPUT

Our first number: 41
Some random number: 4
The first number again: 41
```

MEMORY ALLOCATION FUNCTIONS:

1. malloc():

void * malloc (size_t size);

Size of the memory block in bytes. If the request is successful then a pointer to the memory block is returned. If the function failed to allocate the requested block of memory, a null pointer is returned.

```
#include<stdio.h>
#include<stdlib.h>
int main (){
		int * buffer;
		buffer = (int*) malloc (10*sizeof(int));
		if (buffer==NULL) {
				printf("Error allocating memory!");
				exit (1);
		}
		free (buffer);
```

```
                return 0;
}
```

2. realloc():

*void * realloc (void * ptr, size_t size);*

The function realloc() reallocates a memory block with a specific new size. If you call realloc() the size of the memory block pointed to by the pointer is changed to the given size in bytes. This way you are able to expand and reduce the amount of memory you want to use (if available of course.)

It is possible that the function moves the memory block to a new location, in which way the function will return this new location. If the size of the requested block is larger then the previous block then the value of the new portion is indeterminate.

If the pointer is NULL then the function will behave exactly like the function malloc(). It will assign a new block of a size in bytes and will return a pointer to it.

If the size is 0 then the memory that was previously allocated is freed as if a call of the function free() was given. It will return a NULL pointer in that case.

```
#include<stdio.h>
#include<stdlib.h>
int main ()
{
        int * buffer;
        /*get a initial memory block*/
        buffer = (int*) malloc (10*sizeof(int));
        if (buffer==NULL)    {
                printf("Error allocating memory!");
                exit (1);
        }
        /*get more memory with realloc*/
        buffer = (int*) realloc (buffer, 20*sizeof(int));
        if (buffer==NULL)    {
                printf("Error reallocating memory!");
                //Free the initial memory block.
                free (buffer);
                exit (1);
        }
        free (buffer);
        return 0;
}
```

3. calloc():

*void * calloc (*size_t nmemb *, size_t size);*

The function calloc() will allocate a block of memory for an array. All of these elements are size bytes long. During the initialization all bits are set to zero.

```
#include <stdlib.h>
#include <stdio.h>
main(){
   unsigned num;
   int *ptr;
   printf("Enter the number of type int to allocate: ");
   scanf("%d", &num);
   ptr = (int*)calloc(num, sizeof(int));
   if (ptr != NULL)
      puts("Memory allocation was successful.");
   else
      puts("Memory allocation failed.");
   return(0);
 }
```

```
Enter the number of type int to allocate: 100
Memory allocation was successful.
```

```
Enter the number of type int to allocate: 99999999
Memory allocation failed.
```

WHAT IS FUNCTION?

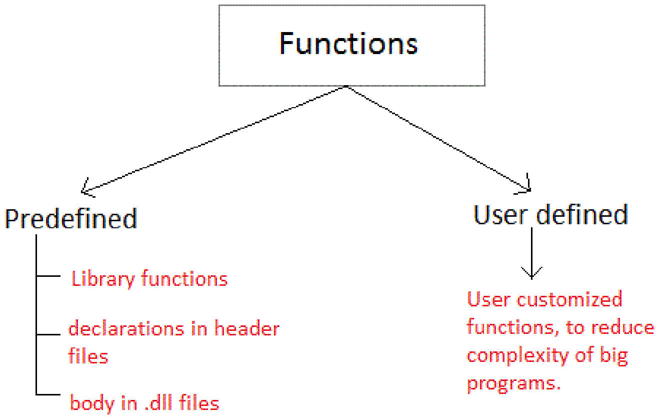
A ***function*** is an independent self contained block of statement that performs a specific task and optionally returns a value to the calling program.

A function is a block of code that performs a particular task.

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

These functions defined by the user are also know as User-defined Functions C functions can be classified into two categories,

1. Standared Library functions
2. User-defined functions


Functions
Predefined
User defined
Library functions
declarations in header files
body in .dll files
User customized functions, to reduce complexity of big programs.

Standard library(Predefined) functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

User defined function

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main ()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows

```
return_type function_name(parameter list)
 {
        body of the function
 }
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** - A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Functions Name** - This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** - A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** - The function body contains a collection of statements that define what the function does.

User-defined function Example

Add two integers. To perform this task, an user-defined function addNumbers() is defined.

Main function	User Defined Function

#include <stdio.h>
int addNumbers(int a, int b);   **// function prototype**
void 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;
}
	// function definition
	int addNumbers(int a,int b)
	{
	   int result;
	   result = a+b;
	   return result; **// return statement**
	}

1. Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax: - **returnType functionName(type1 argument1, type2 argument2,...);**

In the above example, **int addNumbers(int a, int b);** is the function prototype which provides following information to the compiler:

- name of the function is addNumbers()
- return type of the function is int
- two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

2. Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax : **Function_Name(argument1, argument2, ...);**

In the above example, function call is made using **addNumbers(n1,n2);** statement inside the main().

Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

Syntax

```
ReturnType Function_Name (type1 argument1, type2 argument2, ...)
{
   //body of the function
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

3. Passing arguments to a function

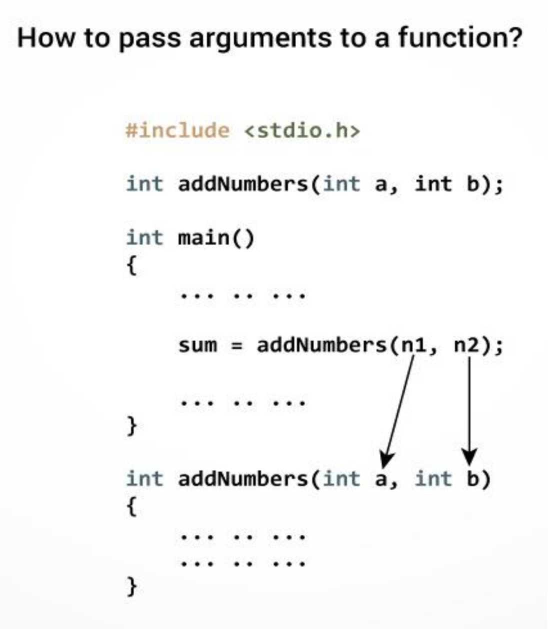
In programming, argument refers to the variable passed to the function. In the above example, two variables **n1 and n2** are passed during function call.

The parameters **a and b** accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.

If n1 is of char type, a also should be of char type. If n2 is of float type, variable b also should be of float type.

A function can also be called without passing an argument.



How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

4. Return Statement

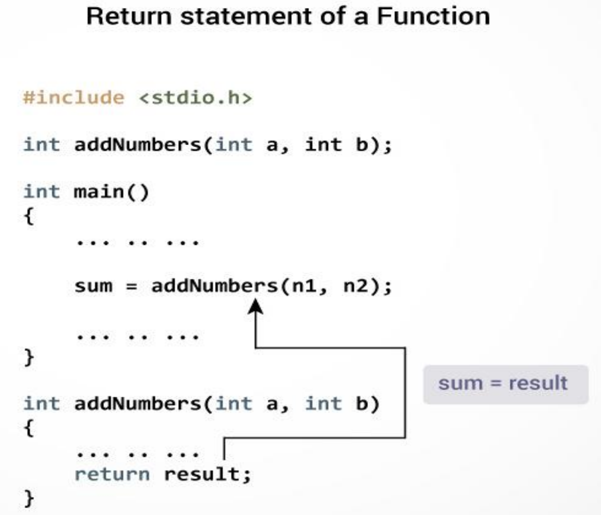
The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

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

Syntax return (expression);

Example

```
return a;
return (a+b);
```



Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

The type of value returned from the function and the return type specified in function prototype and function definition must match.

void

Generally void is an empty or incomplete data type means "nothing" or "no type". You can think of void as absent and it used as a return type of functions that return no value.
Example:-

```
void  func(int n)     //  It indicates that the function returns no result.
int func(void)        //  It indicates that the function has no parameters.
```

When declare a pointer variable with a specific data type that cannot hold the other data type variable's address. It is invalid and causes compilation error.
Example:-

```
char  *p;
int  v1;
p=&v1;           // This is invalid because ‘p’ is a character pointer variable.
```

In C to overcome this problem, void is used to declare the pointer.

Eample :-

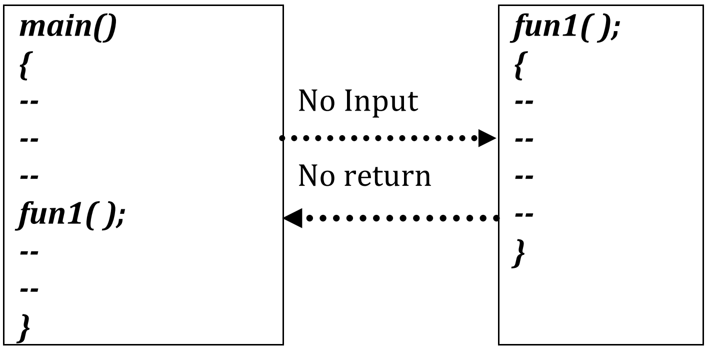
```
void *p;         // Now p is a general purpose pointer variable
```

When a pointer variable is declared with keyword void it turns into a general purpose pointer variable. **Now the address of any variable of any data type such as char, int, and float can be assigned to a void pointer variable.**

TYPE OF CALLING FUNCTION

1 FUNCTIONS WITH NO ARGUMENT, NO RETURN VALUE:

if the function has no arguments, it does not receive any data from the calling function. Similarly, if it does not return a value, the calling function does not receive any data from the called function. Means there is no data transfer between the calling function and the called function.Here the dotted line indicates there is no data transfer but only a transfer of control.



```
#include<stdio.h>
void printline(void);
void value(void);
void main()
{
        printline();      // Call UDF1
        value();           //Call UDF2
       printline();      // Call UDF1
}
```

```
// UDF 1
void printline() {
     int i;
     for(i=1;i<=35;i++)
               printf(“%c”,’-‘);
     printf(“\n”);
}
// UDF 2
void  value() {
          printf(“C Language”);
}
```

Output:

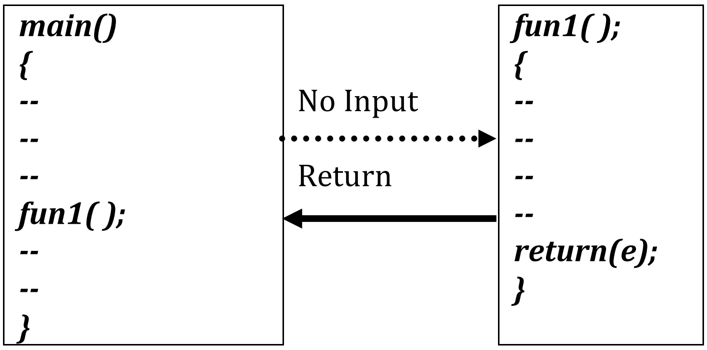
```
----------------------------
C Language
----------------------------
```

When there is nothing to be returned, the return statement

2 FUNCTIONS WITH NO ARGUMENT, With RETURN VALUE:

if the function has no arguments, it does not receive any data from the calling function. But if called function returns a value to the caller function then it is called function with no arguments but return value.

Here the dotted line indicates there is no data transfer from caller function to called function but called function transfer data to the calling function.



#include<stdio.h>
int sqtr1(void);
main(){
int a;
a=sqrt1();
printf("ans:%d",a);
}
	// User Define Function
	int sqrt1(void){
	int x;
	printf("enter number");
	scanf("%d",&x);
	return(x*x);
	}

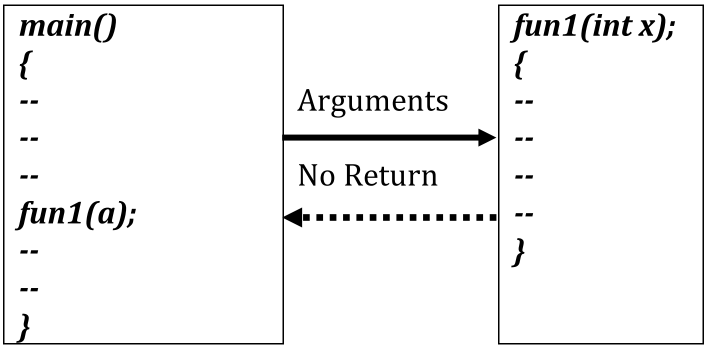
Output: enter number:5 ans:25	

Here we do not pass any value to function but we receive data from the function so it is called no arguments but function returns a value.

3 FUNCTIONS WITH ARGUMENT, NO RETURN VALUE:

When the arguments pass but if function does send any return value then it is called function with arguments but no return value.

The data communication between the calling function and the called function with the arguments but no return value is shown below:



The arguments used at the time of function definition are known as formal arguments. The call to such function needs same number of arguments to be passed to it at the time of function call. These arguments are known as actual arguments. The formal arguments and the actual arguments should have one to one relation. The formal arguments and the actual arguments should match in number, type and order.

EXAMPLE:

```
#include<stdio.h>
void mul(int, int);
main()
{
        int a,b;
        printf("enter two numbers:);
        scanf("%d%d",&a,&b);
       mul(a,b);
}
```

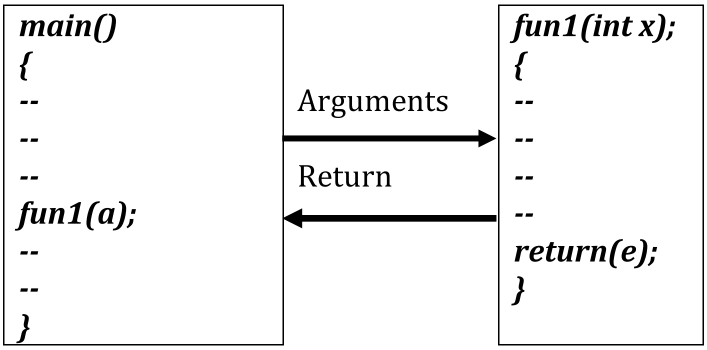
```
// User Define Function

void mul(int x, int y)
{
int m;
m=x*y;
printf("mul is :%d",m);
}
```

```
Output:
enter two numbers:5 4
mul is:20
```

Here function pass two arguments but does receive any return value. Mul() pass two integer types arguments mul(a,b) this called actual arguments where as mul( ) receive two arguments mul(int x,int y) this is called formal arguments. After multiplication of "x" and "y" it assigns value to "m" and then "m" is printed as output result of two integers.

4 FUNCTIONS WITH ARGUMENT, WITH RETURN VALUE:

If the called function receive data from the calling function trough arguments, and send back any value to the calling function then it is called function with arguments and with return value. Receive data back we may use it in the calling function for further processing. Here two data communication between calling and called functions.



The program given here is used to find factorial of a number. For example if we need to find factorial of 5, then it is represented as **5! = 5 * 4 * 3 * 2 * 1**

EXAMPLE:

```
#include<stdio.h>
int factorial(int);
void main()
{
        int n,f;
        clrscr();
        printf("enter a number:);
        scanf("%d",&n);
        f=factorial(n);
        printf("factorial of %d is :%d",n,f);
}
```

```
int factorial(int num)
{
        int i,r=1;
        for(i=1;i<=num;i++)
                r=r*i;
       return(r);
}
```

```
Output:
enter a number:5
factorial of 5 is :120
```

Advantages of using functions:

1. Dividing program into functions makes understanding and maintenance of program more convenient.
2. Functions written once in a program can be used many times. This reduces the length of the program and thus saves memory.
3. Functions written in one program can be used in other programs also.
   Now let us see how to use a function in program. In diagram shows control flow of a program that uses function.

Function CALL BY VALUE:

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

```
# include< stdio.h >
void main()
{
     int a=10,b=20;
    printf("\n Value of a=%d  b=%d",a,b);
    fncn(a,b);
}
```

```
fncn(int   x, int  y)
{
    int t;
    t=x;
    x=y;
    y=t;
    printf("\n Value of x=%d  y=%d",a,b);
}
```

```
Value of x=20  y=10
Value of  a=10 b=20
```

Function CALL BY REFERENCE:

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

```
# include< stdio.h >
void main()
{
          int a=10,b=20;
         printf("Before function call");
         printf("\n Value of a and b =%d %d",a,b);
         fncn(&a,&b);
```

```
fncn(int *x, int *y)
{
        int t;
        t=*x;
        *x=*y;
        *y=t;
}
```

```
        printf(“after function call”);
        printf(“\n Value of a and b =%d %d”,a,b);
}
```

```
Value of a and b before function call =10   20
Value of a and b after function call =20    10
```

CALL BY VALUE	CALL BY REFERENCE
While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”.	While calling a function, instead of passing the values of variables, we pass address of variables (location of variables) to the function known as “Call By References.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the addresses of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passes by Simple technique.	Pointer variables are necessary to define to store the address values of variables.

RECURSION:

The term *recursion* refers to a situation in which a function calls itself either directly or indirectly. *Indirect recursion* occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int fact,a;
    clrscr();
    printf(“enter any number “);
    scanf(“%d:,&a);
    fact=rec(a);
   printf(“factorial value is  =%d”,fact);
}
```

```
rec(int x){
        int f;
        if(x==1)
        {
                return(1);
        }
        else
        {
                f=x*rec(x-1);
                return(f);
        }
}
```

```
enter any number 5
factorial value is =120
```

Our recursive function, factorial(), value passed is assigned to a. the value of a is checked. If it's 1, the program returns the value of 1. If the value isn't 1, a is set equal to itself times the value of factorial (a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) isn't equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement is true. If the value of the factorial is 3, the factorial is evaluated as 3 * (3-1) * ((3-1)-1)

STORAGE CLASSES & SCOPE OF VARIABLE:

What is a Storage Class?

Storage class in C decides the part of storage to allocate memory for a variable, it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable. There are four storage classes in C those are *automatic*, *register*, *static*, and *external*.

Storage Class Specifiers

There are four storage class specifiers in C as follows, typedef specifier does not reserve storage and is called a storage class specifier only for syntactic convenience. It is not a storage class specifier in the common meaning.

- auto
- register
- extern
- static
- typedef

These specifiers tell the compiler how to store the subsequent variable. The general form of a variable declaration that uses a storage class is shown here:

storage_class_specifier data_type variable_name;

At most one storage class specifier may be given in a declaration. If no storage class specifier is specified then following rules are used:

1. Variables declared inside a function are taken to be **auto**.
2. Functions declared within a function are taken to be **extern.**
3. Variables and functions declared outside a function are taken to be **static**, with *external linkage*.

Variables and functions having ***external linkage*** are available to all files that constitute a program. File scope variables and functions declared as **static**(described shortly) have *internal linkage*. These are known only within the file in which they are declared. Local variables have no linkage and are therefore known only within their own block.

Types of Storage Classes

There are four storage classes in C they are as follows:

1. Automatic Storage Class
2. Register Storage Class
3. Static Storage Class
4. External Storage Class

1. Automatic variables: auto

Scope: Variable defined with auto storage class is local to the function block inside which they are defined.

Default Initial Value: Any random value i.e garbage value.

Lifetime: Till the end of the function/method block where the variable is defined.

A variable declared inside a function without any storage class specification, is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function's execution is completed. Automatic variables can also be called local variables because they are local to a function. By default they are assigned garbage value by the compiler.

```
#include<stdio.h>
void main()
{
  int detail;
  // or
  auto int details;    //Both are same
}
```

2. External or Global variable

Scope: Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.

Default initial value: 0(zero).

Lifetime: Till the program doesn't finish its execution, you can access global variables.

A variable that is declared outside any function is a Global Variable. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero). One important thing to remember about global variable is that their values can be changed by any function in the program.

```
#include<stdio.h>
int number;     // global variable
void main(){
  number = 10;
  printf("I am in main function. My value is %d\n", number);
  fun1();     //function calling, discussed in next topic
  fun2();     //function calling, discussed in next topic
}
/* This is function 1 */
fun1() {
  number = 20;
  printf("I am in function fun1. My value is %d", number);
}
/* This is function 1 */
fun2() {
  printf("\nI am in function fun2. My value is %d", number);
}
```

```
I am in function main. My value is 10
I am in function fun1. My value is 20
I am in function fun2. My value is 20
```

Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

Declaring the storage class as global or external for all the variables in a program can waste a lot of memory space because these variables have a lifetime till the end of the program. Thus, variables, which are not needed till the end of the program, will still occupy the memory and thus, memory will be wasted.

3. **Static variables**

Scope: Local to the block in which the variable is defined
Default initial value: 0(Zero).
Lifetime: Till the whole program doesn't finish its execution.

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, staticvariable is initialized only once and remains into existence till the end of the program. A staticvariable can either be internal or external depending upon the place of declaration. Scope of internal static variable remains inside the function in which it is defined. External static variables remain restricted to scope of file in which they are declared.

They are assigned 0 (zero) as default value by the compiler.

Main function	UDF function	Output

#include<stdio.h>
void test();
//Function declaration
// (discussed in next topic)
int main(){
   test();
   test();
   test();
}
	void test(){
	   static int a = 0; //a static variable
	   a = a + 1;
	   printf("%d\t",a);
	}
		1 2 3

4. **Register variable**

Scope: Local to the function in which it is declared.
Default initial value: Any random value i.e garbage value
Lifetime: Till the end of function/method block, in which the variable is defined.

Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers. One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time.

Syntax : register int number;

Note: Even though we have declared the storage class of our variable number as register, we cannot surely say that the value of the variable would be stored in a register. This is because the number of registers in a CPU is limited. Also, CPU registers are meant to do a lot of important work. Thus, sometimes they may not be free. In such scenario, the variable works as if its storage class is auto.

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

Which storage class should be used and when

To improve the speed of execution of the program and to carefully use the memory space occupied by the variables, following points should be kept in mind while using storage classes:

- We should use static storage class only when we want the value of the variable to remain same every time we call it using different function calls.
- We should use register storage class only for those variables that are used in our program very oftenly. CPU registers are limited and thus should be used carefully.
- We should use external or global storage class only for those variables that are being used by almost all the functions in the program.
- If we do not have the purpose of any of the above mentioned storage classes, then we should use the automatic storage class.

UNIT 4

1. ARRAY

WHAT IS ARRAY?

We can use normal variables (v1, v2, v3...) when we have small number of objects, but if we want to store large number of instances, it becomes difficult to manage them with normal variables. The idea of array is to represent many instances in one variable.

Array is a collection or group of similar data type elements stored in contiguous memory.

OR

An array is a collection of data items, all of the same type, accessed using a common name.

Arrays are reffered to as structured data types. An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations.

Here the words,

- **Finite** means data range must be defined.
- **Ordered** means data must be stored in continuous memory addresses.
- **Homogenous** means data must be of similar data type.

Types of ARRAY:

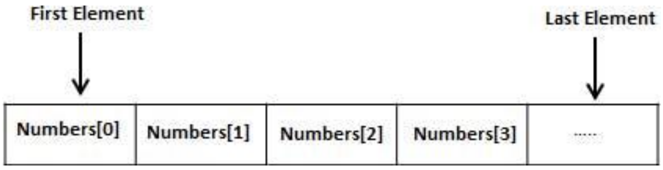
1. Single-Dimensional arrays(1D)
   I. Numeric (int, float, double) *II.* String (Char)
2. Two Dimensional Arrays(2D)
   I. Numeric (int, float, double) *II.* String (Char)
3. Multi Dimensional Arrays
   I. Numeric *(int, float, double)*

Single-dimensional arrays

1. Numeric single Dimensional Array

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

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



Declaring Arrays

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

SYNTEX: Data_type ArrayName [ArraySize];

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and type can be any valid C data type.

For example, to declare a 10-element array called balance of type int, use this statement

Example: int balance [10];

ARRAY[INDEX]	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
MEMORY OF ELEMENT	2 Byte	2 Byte	2 Byte	2 Byte	2 Byte	2 Byte	2 Byte	2 Byte	2 Byte	2 Byte
MEMORY ADDDRESS	2001	2003	2005	2007	2009	2011	2013	2015	2017	2019

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

Initializing Arrays

I. Compile Time Array

- Size is Specified Directly
- Size is Specified Indirectly

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

Example: int balance[5] = {100, 200, 300, 400, 500}; // Size is Specified Directly

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

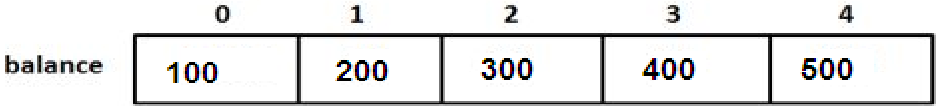
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write

Example: int balance[] = {100, 200, 300, 400, 500}; // Size is Specified Indirectly

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

balance[4] = 500;

The above statement assigns the 5th element in the array with a value of 500. All arrays have **0 as the index of their first element** which is also **called the base index** and the last index of an array will be **total size of the array minus 1**. Shown below is the pictorial representation of the array we discussed above –



II. Run time 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
        }
        printf("Value in Array:");
        for(j=0; j<4; j++)  {
                printf("%d \n",arr[j]);
        }
  getch();
}
```

Accessing Array Elements

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

Example: **int salary = balance[9];**

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

```
#include <stdio.h>
void main ()
{
        int n[ 10 ];                    /*  n is an array of 10 integers   */
        int i,j;
        for ( i = 0; i < 5; i++ )       /*  initialize elements of array n to 0  */
        {
                    n[ i ] = i + 100;     /*  set element at location i to i + 100  */
         }
        for (j = 0; j < 5; j++ )        /*  output each array element's value  */
        {
                printf("Element[%d] = %d\n", j, n[j] );
        }
}
```

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

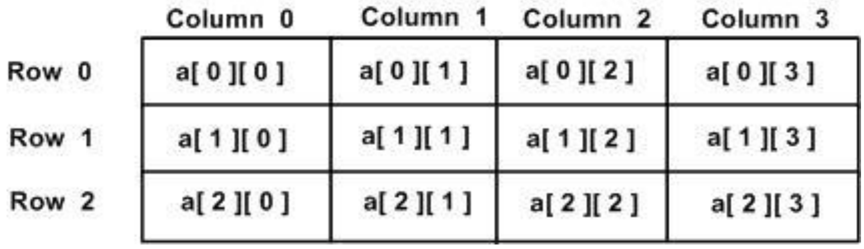
```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
```

Two-Dimensional Arrays(2D)

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows

SYNTEX: : DataType ArrayName [X][Y];

Where type can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be considered as a table which will have **x number of rows** and **y number of columns**. A two-dimensional array, which contains three rows and four columns, can be shown



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array a is identified by an element name of the form **a[i][j]**, where **'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'**.

Initializing Two-Dimensional Arrays

Two dimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
  {0, 1, 2, 3} ,        /*  initializers for row indexed by 0 */
  {4, 5, 6, 7} ,        /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}        /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional.

Example : int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

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: int val = a[2][3];

The above statement will take the **4th element from the 3rd row of the array**. You can verify it in the above figure.

Example	OUTPUT

#include <stdio.h>
void main () {
    /* an array with 5 rows and 2 columns*/
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
    int i, j;
   /* output each array element's value */
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 2; j++) {
            printf("a[%d][%d] = %d\n", i,j, a[i][j]);
        }
    }
}
	a[0][0]: 0
	a[0][1]: 0
	a[1][0]: 1
	a[1][1]: 2
	a[2][0]: 2
	a[2][1]: 4
	a[3][0]: 3
	a[3][1]: 6
	a[4][0]: 4
	a[4][1]: 8

When the above code is compiled and executed, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

ARRAY PASS TO FUNCTION

Array elements can be passed to a function by calling the function by value or by reference.

- **Call by value, we pass values of array elements to the function**
- **Call by reference. we pass addresses of the array elements to the function.**

PASSING SINGLE DIMENSIONAL ARRAY TO FUNCTION:

1. Call By Value

Here, display ( ) function is used to print out the array elements. Note that here we first write function prototype and there are two arguments one for array and another for total number of elements. If we do not pass the total number of elements then function would not know when to terminate the for loop.

Main function	UDF for call by Value

**#include<stdio.h>
#include<conio.h>
void display(int [] ,int);
void main()
{
int marks[ ]={55,65,75,56,78,78,90};
clrscr();
display(marks,7);   // call Function
getch();
}**
	**void display(int m[],int n)
	{
	int i;
	for(i=0;i<n-1;i++)
	printf("%d,",m[i]);
	}**

Output is: 55,65,75,56,78,78,90,	

2. **Call By references**

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointeras a parameter to receive the passed address.

Main function	UDF for call by References

**#include<stdio.h>
#include<conio.h>
void display(int [] ,int);
void main() {
int marks[ ]={55,65,75,56,78,78,90};
clrscr();
display(&marks,7);   // call Function
getch();
}**
	**void display(int *m,int n)
	{
	int i;
	for(i=0;i<n-1;i++)
	printf("%d,",*m);
	m++;
	}**

Output is: 55,65,75,56,78,78,90,	

PASSING TWO DIMENSIONAL ARRAY TO FUNCTION:

1. **Call By Value**

Main function	UDF for call by Value

**#include<stdio.h>
#include<conio.h>
void show(int[3][3],int,int);
void main()
{
int i,j;
int [3][3]={10,20,25,40,38,60,70,80,90};
clrscr();
show(a,3,3);
getch();
}**
	**void show(int x[3][3],int row,int col)
	{
	int i,j;
	for(i=0;i<row;i++) {
	for(j=0;j<col;j++) {
	printf("\t%d",x[i][j]);
	}
	printf("\n");
	}
	}**

```
Output:
10 20 25
40 38 60
70 80 90
```

There are show methods to pass two dimensional arrays to function. We can also pass values of array elements to the function. Here show function has three arguments first is for array name and remaining two are for array elements size. Using two for loops in function we can access elements of arrays row by row and output will be displayed.

2. Call By references

Main function

```
#include<stdio.h>
#include<conio.h>
void show(int[3][3],int,int);
void main()
{
   int i,j;
   int [3][3]={10,20,25,40,38,60,70,80,90};
   clrscr();
   show(&a,3,3);
   getch();
}
```

UDF for call by Reference

```
void show(int *x,int row,int col)
{
    int i,j;
    for(i=0;i<row;i++)  {
         for(j=0;j<col;j++)  {
                 printf("\t%d",*x);
                 x++;
         }
         printf("\n");
     }
}
```

```
Output:
10 20 25
40 38 60
70 80 90
```

There are different methods to pass two dimensional arrays to function. We can pass the arrays using reference to the function. Here show function has three arguments, first is address of array and remaining two are for array elements row size and column size. Using two for loops in function we can access elements of arrays row by row and column by column. Output will be displayed.

STRING

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A string is actually one-dimensional array of characters in C language

1. Single Dimension Array (String) Declaration

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

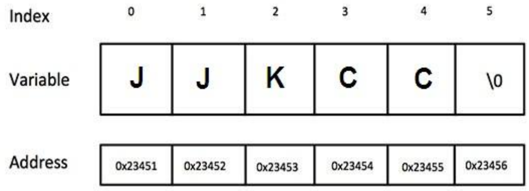
Syntax: char str_name[size];

In the above syntax str_name is any name given to the string variable and size is used define the length of the string, i.e the number of characters strings will store. Please keep in mind that there is an extra terminating character which is the **Null character ('\0') used to indicate termination of string** which differ strings from normal character arrays.

Initializing a String

A string can be initialized in different ways. We will explain this with the help of an example. Below is an example to declare a string with name as str and initialize it with "JJKCC".

- char str[] = "JJKCC";
- char str[5] = "JJKCC";
- char str[] = {'J','J','K','C','C','\0'};
- char str[5] ={'J','J','K','C','C','\0'};



Index	0	1	2	3	4	5
Variable	J	J	K	C	C	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Let us now look at a sample program to get a clear understanding of declaring and initializing a string in C and also how to print a string.

EXAMPLE	OUTPUT

#include<stdio.h>
void main() {
char str[] = "JJKCC";   // declare and initialize string
printf("%s",str);   //   print string
}
	JJKCC

We can see in the above program that strings can be printed using a normal printf statements just like we print any other variable. Unlike arrays we do not need to print a string, character by character. The C language does not provide an inbuilt data type for strings but it has an access specifier “%s” which can be used to directly print and read strings. Sample program to read a string from user

EXAMPLE	OUTPUT

#include<stdio.h>
void main() {
char str[50];   // declaring string
scanf("%s",str);   // reading string
printf("%s",str);   // print string
}	

You can see in the above program that string can also be read using a single scanf statement. We know that the ‘&’ sign is used to provide the address of the variable to the scanf () function to store the value read in memory. As str[] is a character array so using str without braces ‘[‘ and ‘]’ will give the base address of this string. That’s why we have not used **‘&’** in this case as we are already providing the base address of the string to scanf.

Passing strings to function

As strings are character arrays, so we can pass strings to function in a same way we pass an array to a function. Below is a sample program to do this:

Main function	UDF	OUTPUT

#include<stdio.h>
void main() {
// declare and initialize string
char str[] = "JJKCC";
// print string by passing string
// to a different function
printStr(str);
}
	void printStr(char str[])
	{
	printf("String is : %s",str);
	}
		String is : JJKCC

TWO-DIMENSIONAL ARRAY OF CHARACTERS

We know that string is nothing but an array of characters which ends with a ‘\0’. 2D character arrays are very similar to the 2D integer arrays. We store the elements and perform other operations in a similar manner. A 2D character array is more like a String array. It allows us to store multiple strings under the same name.

DECLARATION 2D CHARACTER ARRAY

A **2D character array** is declared in the following manner

Syntax: char names [5][10];

The order of the subscripts is too kept in mind during declaration. The **first** subscript[5] represents the **number of Strings (ROW)** that we want our array to contain and the **second** subscript [10] represents the **length of each String(Column)**. This is static memory allocation. We are giving **5*10=50** memory locations for the array elements to be stored in the array.

INITIALIZATION OF 2D CHARACTER ARRAY

```
char names[5][10]={ “akshay”, ”parag”, ”raman“, ”kishan”, ”gopal” };
```

Here, we can initialize the array with 5 strings values, **each with maximum 10 characters long**. If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory. See the diagram below to understand how the elements are stored in the memory location

	Memory Location	Array Elements									
ROW INDEX	(Base Address)	Length of Each String is 10									
		COLUMN INDEX									
		0	1	2	3	4	5	6	7	8	9
0	65454	a	k	s	h	a	y	\0			
1	65464	p	a	r	a	g	\0				
2	65474	r	a	m	a	n	\0				
3	65484	k	i	s	h	a	n	\0			
4	65494	g	o	p	a	l	\0				

The areas marked in blank shows the memory locations that are reserved for the array but are not used by the string. Each **character occupies 1 byte of storage** from the memory.

TAKING DATA INPUT FROM USER

In order to take string data input from the user we need to follow the following syntax:

```
for(i=0 ;i<5 ;i++ )
   scanf("%s",&name[i][0]);
```

Here we see that the second subscript remains [0]. This is because it shows the length of the string and before entering any string the length of the string is 0.

PRINTING THE ARRAY ELEMENTS

The way a 2D character array is printed in not the same as a 2D integer array. This is because we see that all the spaces in the array are not occupied by the string entered by the user. If we display it in the same way as a 2D integer array we will get unnecessary garbage values in unoccupied spaces. Here is how we can display all the string elements:

```
for(i=0 ;i<5 ;i++)
      printf("%s\n",name[i]);
```

This format will print only the string contained in the index numbers specified and eliminate any garbage values after '\0'.

Passing strings to function

As strings are character arrays, so we can pass strings to function in a same way we pass an array to a function. Below is a sample program to do this:

Main function	UDF	OUTPUT

#include<stdio.h>
void main() {
// declare and initialize string
char names[5][10]={
“akshay”,
”parag”,
”raman“,
”kishan”,
”gopal”
};
// print string by passing string
// to a different function
For(i=0;i<=4; i++)
printStr(name[i]);
}
	void printStr(char str[])
	{
	printf("\nString is : %s",str);
	}
		String is: **akshay**
		String is: **parag**
		String is: **raman**
		String is: **kishan**
		String is: **gopal**

Multidimensional Arrays

An Array having more than one dimension is called Multi Dimensional array in C. we discussed about Two Dimensional Array which is the simplest form of C Multi Dimensional Array.

In C Programming Language, by placing n number of brackets [ ] we can declare n-dimensional array where n is dimension number.

Example,

- int a[2][3][4] Three Dimensional Array
- int a[2][2][3][4] Four Dimensional Array

we will explain about the Three Dimensional Array in C for better understanding. You can try 4D array on your own.

Syntax : Data_Type Array_Name[Tables][Row_Size][Column_Size]

- **Data_type:** It will decide the type of elements it will accept. For example, If we want to store integer values then we declare the Data Type as int, If we want to store Float values then we declare the Data Type as float etc
- **Array_Name:** This is the name you want to give it to array.
- **Tables:** It will decide the number of tables an array can accept. Two Dimensional Array is always single table with rows and columns. Whereas Multi Dimensional array in C is more than 1 table with rows and columns.
- **Row_Size:** Number of Row elements an array can store. For example, Row_Size =10 then array will have 10 rows.
- **Column_Size:** Number of Column elements an array can store. For example, Column_Size = 8 then array will have 8 Columns.

We can calculate the maximum number of elements in a Three Dimensional using: [Tables] * [Row_Size] * [Column_Size]

Example : int Employees[2][4][3];

1. Here, we used int as the data type to declare an array. So, above array will accept only integers. If you try to add float values then it will through an error.
2. Employees is the array name
3. Number of tables = 2 so, this array will hold maximum 2 levels of data (rows and columns).
4. The Row size of an Array is 4 it means, Employees array will only accept 4 integer values as rows.
    - If we try to store more than 4 then it will throw an error.
    - We can store less than 4. For Example, If we store 2 integer values then remaining 2 values will be assigned to default value (Which is 0).
5. The Column size of an Array is 3 it means, Employees array will only accept 3 integer values as columns.
    - If we try to store more than 3 then it will throw an error.
    - We can store less than 3. For Example, If we store 1 integer values then remaining 2 values will be assigned to default value (Which is 0).
6. Finally, Employees array can hold maximum of 24 integer values (2 * 4 * 3 = 24).

Multi Dimensional Array Initialization

We can initialize the C Multi Dimensional Array in multiple ways

<u>First Approach of Multi Dimensional Array</u>

```
Example: int Employees[2][4][3] = {
                    { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },
                    { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                  };
```

Here, We have 2 tables and 1st table holds 4 Rows * 3 Columns and 2nd table also holds 4 Rows * 3 Columns

First three elements of the first table will be 1st row, second three elements will be 2nd row, next 3 elements will be 3rd row and last 3 element will be 4th row. Here we divided them into 3 because our column size = 3 and We surrounded each row with curly braces ({}). It is always good practice top use the curly braces to separate the rows.

<u>Second Approach of Multi Dimensional Array</u>

```
Example : int Employees[2][ ][3] = {
                    { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },
                    { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                    };
```

Here, We did not mention the row size but the compiler is intelligent enough to calculate the size by checking the number of elements inside the row.

We can also write

```
Eample: int Employees[2][4][ ] = {
                    { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },
                    { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                   };
```

<u>Third Approach for Multi Dimensional Array</u>

```
int Employees[2][4][3] = {
                    { { 10 }, {15, 25}, {22, 44, 66}, {33, 55, 77} },
                   { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                 };
```

Here we declared Employees array with row size = 4 and column size = 3 but we only assigned 1 column in the 1st row and 2 columns in the 2nd row of the first table. In these situations, the remaining values will be assigned to default values (0 in this case).
The above array will be:

```
int Employees[2][4][3] = {
                          { {10, 0, 0}, {15, 25, 0}, {22, 44, 66}, {33, 55, 77} },
                         { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                       };
```

Fourth Approach Multi Dimensional Array

Above 3 ways are good to store small number of elements into the array, What if we want to store 100 rows or 50 column values in 20 tables. It will be a nightmare to add all of them using any of the above mentioned approaches. To resolve this, we can use the loop concept here:

```
int tables, rows, columns, Employees[20][100][100];
for (tables = 0; tables < 20; tables ++) {
 for (rows = 0; rows < 100; rows++)  {
  for (columns =0; columns < 100; columns++)   {
    Employees[tables][rows][columns] = tables + rows + columns ;
  }
 }
}
```

Accessing Multi Dimensional Array in C

We can access the C Multi Dimensional array elements using indexes. Using index we can access or alter/change each and every individual element present in the array separately. Index value starts at 0 and end at n-1 where n is the size of a row or column.
For example, if an Array_name[4][8][5] will stores 8 row elements and 5 column elements in each table where table size = 4. To access 1st value of the 1st table use Array_name[0][0][0], to access 2nd row 3rd column value of the 3rd table then use Array_name[2][1][2] and to access the 8th row 5th column of the last table (4thtable) then use Array_name[3][7][4].

Lets see the example of C Multi Dimensional Array for better understanding:

```
int Employees[2][4][3] = { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },
                  { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                 };
```

//To Access the values in the Employees[2][4][3] array

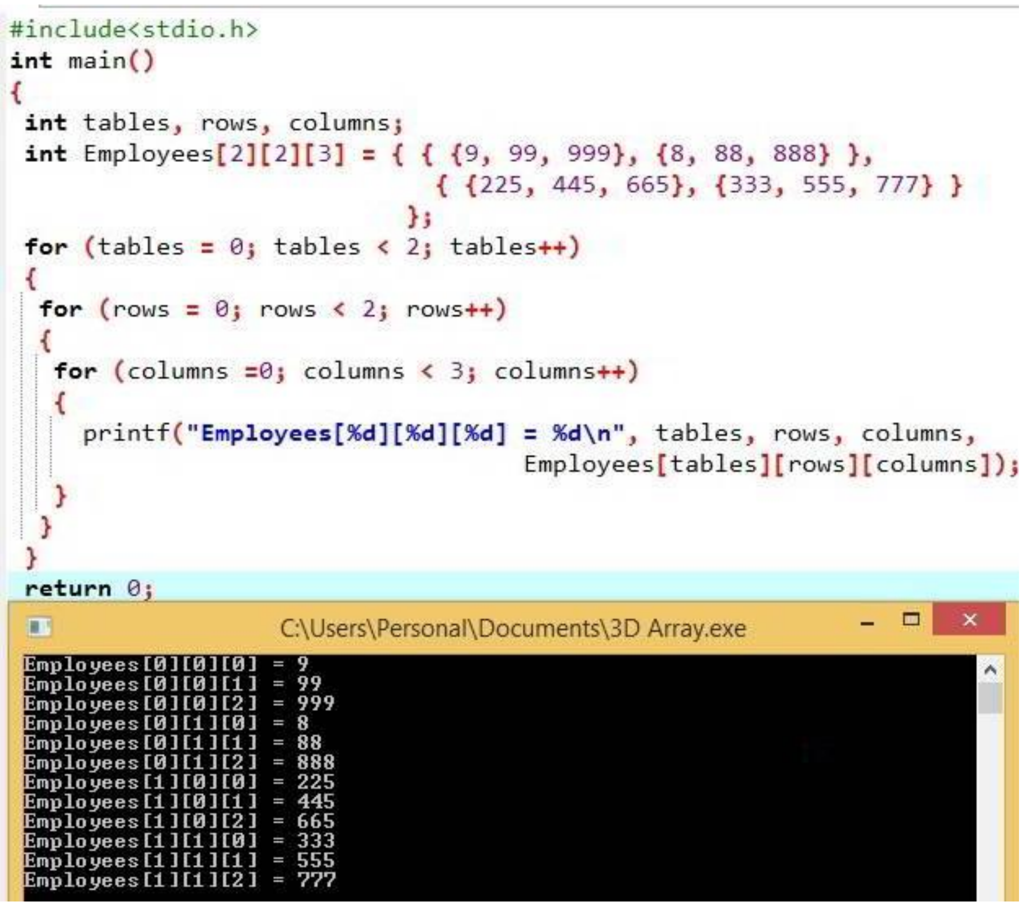
//Accessing First Table Rows & Columns	//Accessing Second Table Rows & Columns
printf("%d", Employees[0][0][0]) = 10 printf("%d", Employees[0][0][1]) = 20 printf("%d", Employees[0][0][2]) = 30 printf("%d", Employees[0][1][0]) = 15 printf("%d", Employees[0][1][1]) = 25 printf("%d", Employees[0][1][2]) = 35 printf("%d", Employees[0][2][0]) = 22 printf("%d", Employees[0][2][1]) = 44 printf("%d", Employees[0][2][2]) = 66 printf("%d", Employees[0][3][0]) = 33 printf("%d", Employees[0][3][1]) = 55 printf("%d", Employees[0][3][2]) = 77	printf("%d", Employees[1][0][0]) = 1 printf("%d", Employees[1][0][1]) = 2 printf("%d", Employees[1][0][2]) = 3 printf("%d", Employees[1][1][0]) = 5 printf("%d", Employees[1][1][1]) = 6 printf("%d", Employees[1][1][2]) = 7 printf("%d", Employees[1][2][0]) = 2 printf("%d", Employees[1][2][1]) = 4 printf("%d", Employees[1][2][2]) = 6 printf("%d", Employees[1][3][0]) = 3 printf("%d", Employees[1][3][1]) = 5 printf("%d", Employees[1][3][2]) = 7

//To Alter the values in the Employees[4][3] array
// It will change the value of Employees[0][2][1] from 44 to 98
Employees[0][2][1] = 98;
// It will change the value of Employees[1][2][2] from 6 to 107
Employees[1][2][2] = 107;

For the large number of rows and columns we can access them using For Loop. Say for example, to access array Employees[10][25][60]

```
int tables, rows, columns;
for (tables = 0; tables < 10; tables ++) {
 for (rows = 0; rows < 25; rows++)   {
   for (columns =0; columns < 60; columns++)    {
          printf("%d", Employees[tables][rows][columns]);
   }
  }
 }
```

Multi Dimensional Array Example



```
#include<stdio.h>
int main()
{
 int tables, rows, columns;
 int Employees[2][2][3] = { { {9, 99, 999}, {8, 88, 888} },
                            { {225, 445, 665}, {333, 555, 777} }
                          };
 for (tables = 0; tables < 2; tables++)
 {
  for (rows = 0; rows < 2; rows++)
  {
   for (columns =0; columns < 3; columns++)
   {
     printf("Employees[%d][%d][%d] = %d\n", tables, rows, columns,
                                   Employees[tables][rows][columns]);
   }
  }
 }
 return 0;
```

```
Employees[0][0][0] = 9
Employees[0][0][1] = 99
Employees[0][0][2] = 999
Employees[0][1][0] = 8
Employees[0][1][1] = 88
Employees[0][1][2] = 888
Employees[1][0][0] = 225
Employees[1][0][1] = 445
Employees[1][0][2] = 665
Employees[1][1][0] = 333
Employees[1][1][1] = 555
Employees[1][1][2] = 777
```

Here, we will declare Three dimensional array and initialize it with some values. Using the for loop we will display each and every individual values present in the array as per the index.

Let us see the C Multi Dimensional Array program execution in iteration wise Table First Iteration
The value of tables will be 0 and the condition (tables < 2) is True. So, it will enter into second for loop (Row Iteration)

- **Row First Iteration**

The value of row will be 0 and the condition (rows < 2) is True. So, it will enter into third for loop (Column Iteration)

- **Column First Iteration**

The value of column will be 0 and the condition (columns < 2) is True. So, it will start executing the statements inside the loop until the condition fails.

printf("Employees[%d][%d][%d] = %d\n", tables, rows, columns, Employees[tables][rows][columns]);

Employees[tables][rows][columns] = Employees[0][0][0] = 9

- **Column Second Iteration**

The value of column will be 1 and the condition (columns < 3) is True. Since we didn't exit from the inner loop (Columns loop), Row value will be 0

Employees[tables][rows][columns] = Employees[0][0][1] = 99

- **Column 3rd Iteration**

The value of columns will be 2 and the condition (columns < 3) is True. Since we didn't exit from the inner loop (Columns loop), Row value will be 0

Employees[tables][rows][columns] = Employees[0][0][2] = 999

After the increment, the value of columns will be 3 and the condition (columns < 3) will fail. So it will exit from the 3rd for loop.

Now the value of rows will be incremented and starts the second row iteration

- **Row Second Iteration**

The value of row will be 1 and the condition (rows < 2) is True. So, it will enter into second for loop

- **Column First Iteration**

The value of column will be 0 and the condition (columns < 3) is True. So, it will start executing the statements inside the loop until the condition fails.

printf("Employees[%d][%d][%d] = %d\n", tables, rows, columns, Employees[tables][rows][columns]);

Employees[tables][rows][columns] = Employees[0][1][0] = 8

- **Column Second Iteration**

The value of column will be 1 and the condition (columns < 3) is True. Since we didn't exit from the inner loop (Columns loop), Row value will be 0

Employees[tables][rows][columns] = Employees[0][1][1] = 88

- **Column 3rd Iteration**

The value of columns will be 2 and the condition (columns < 3) is True. Since we didn't exit from the inner loop (Columns loop), Row value will be 0

Employees[tables][rows][columns] = Employees[0][1][1] = 888

After the increment the value of columns will be 3 and the condition (columns < 3) will fail. So it will exit from the 3rd for loop.

Now the value of rows will be incremented, it means rows =2. Condition (rows < 2) will fail So, it will exit from the 2nd For Loop.

Now, the value of tables will be incremented to 1, it means

for (tables = 1; tables < 2; tables++)

Condition is True so it will repeat the above iteration with table value 1. Once completed the value of tables will be incremented.

Now tables = 2 and the condition (tables < 2) will fail so, it will exit from the First for loop.

UNIT 4
2. STRUCTURE

What is structure?

Structure is a user-defined data type in C which allows you to combine different data types to store a particular type of record. Structure helps to construct a complex data type in more meaningful way. It is somewhat similar to an Array. The only difference is that ***array is used to store collection of similar data types while structure can store collection of any type of data.***

Structure is used to represent a record. Suppose you want to store record of **Student** which consists of student name, address, roll number and age. You can define a structure to hold this information

Structure is a collection of different data types which are grouped together and each element in a C structure is called member.

OR

A structure is a collection of variables of different data types.

- If you want to access structure members in C, structure variable should be declared.
- Many structure variables can be declared for same structure and memory will be allocated for each separately.
- It is a best practice to initialize a structure to null while declaring, if we don't assign any values to structure members.
- Don't forget the semicolon }; in the ending line.

Definition Structure

Keyword struct is used for creating a structure.

Syntax

```
struct structure_name
{
        data_type member1;
        data_type member2;
        .
        data_type member-n;
};
```

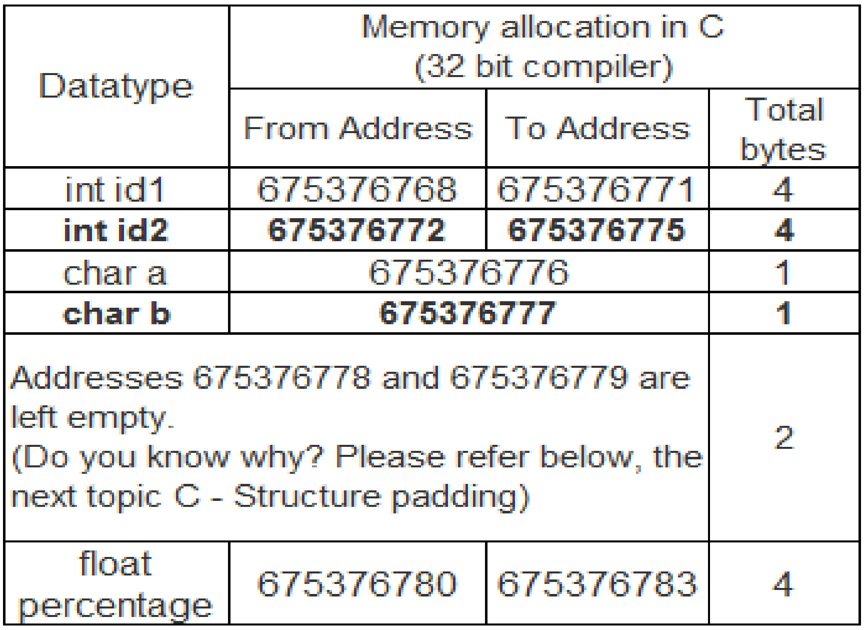
Declaring Structure variables separately	**Declaring Structure Variables with Structure definition**

struct student
{
int id1;
int id2;
char a;
char b;
float per;
};
struct student S1 , S2;
	struct student
	{
	int id1;
	int id2;
	char a;
	char b;
	float per;
	} S1,S2;

Here S1 and S2 are variables of structure student. However this approach is not much recommended	Here S1 and S2 are variables of structure student. However this approach is not much recommended.

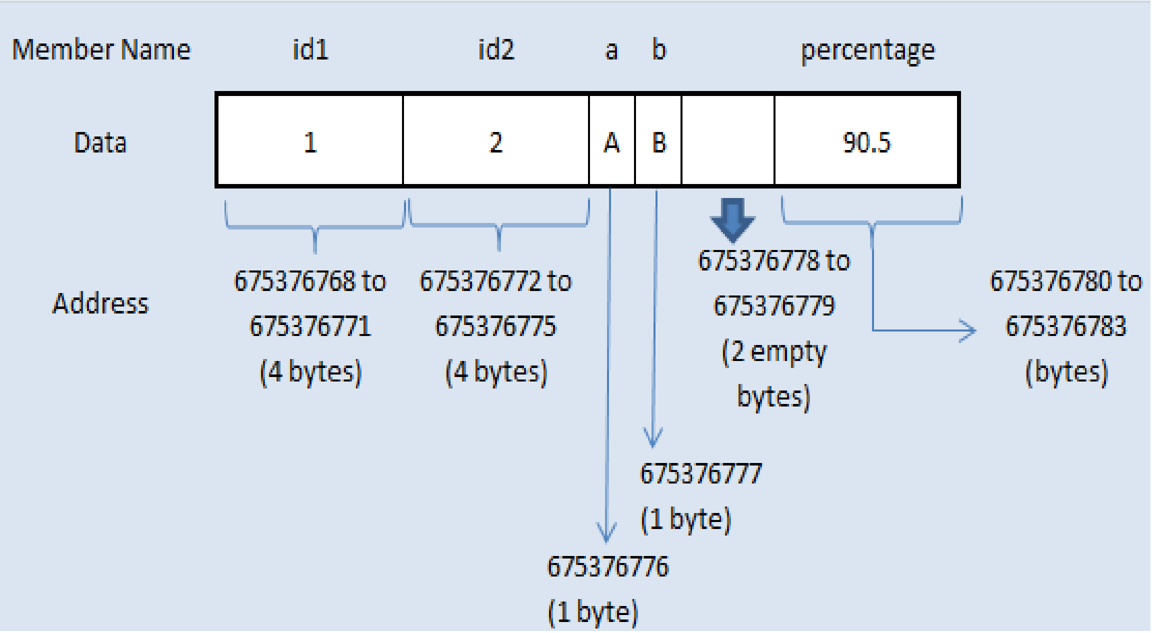
Memory Allocated

- There are 5 members declared for structure in above program. In 32 bit compiler, 4 bytes of memory is occupied by int datatype. 1 byte of memory is occupied by char datatype and 4 bytes of memory is occupied by float datatype.
- Please refer below table to know from where to where memory is allocated for each datatype in contiguous (adjacent) location in memory.



Datatype	Memory allocation in C (32 bit compiler)		
	From Address	To Address	Total bytes
int id1	675376768	675376771	4
int id2	**675376772**	**675376775**	**4**
char a	675376776		1
char b	**675376777**		**1**
Addresses 675376778 and 675376779 are left empty. (Do you know why? Please refer below, the next topic C - Structure padding)			2
float percentage	675376780	675376783	4

- The pictorial representation of above structure memory allocation is given below. This diagram will help you to understand the memory allocation concept in C very easily.


Member Name
id1
id2
a
b
percentage
Data
1
2
A
B
90.5
Address
675376768 to 675376771 (4 bytes)
675376772 to 675376775 (4 bytes)
675376778 to 675376779 (2 empty bytes)
675376780 to 675376783 (bytes)
675376777 (1 byte)
675376776 (1 byte)

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

1. Member operator (.)   // Simple structure Veriable
2. Structure pointer operator(->)   // Pointer structure Veriable

Suppose, we want to access id1 for variable s1. Then, it can be accessed as s1.id1 member operator and s1->id1 as structure pointer operator

ARRAY WITH STRUCTURE:

Usually a program needs to work with more than one instance of the data. For example, in a program to maintain a list of phone numbers, you can define a structure to hold each person's name and number:

```
struct entry
{
  char fname[10];
  char fname[10];
  char phone[8];
 };
```

A phone list must hold many entries, however, so a single instance of the entry structure isn't of much use. What you need is an array of structures of type entry. After the structure has been defined, you can declare an array as follows:

struct entry list[1000];

This statement declares an array named list that contains 1,000 elements. Each element is a structure of type entry and is identified by subscript like other array element types. Each of these structures has three elements, each of which is an array of type char. This entire complex creation is diagrammed in when you have declared the array of structures; you can manipulate the data in many ways.

EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
struct entry{
        char fname[20];
        char lname[20];
        char phone[10];
};
void main(){
    struct entry list[4];
     int i;
     clrscr();
     for(i=0;i<4;i++) {
                printf("\nEnter First Name  :");
                scanf("%s",list[i].fname);
                printf("\nEnter last name   :");
                scanf("%s",list[i].lname);
                printf("\nEnter Phone in 123-4567 Format :");
                scanf("%s",list[i].phone);
        }
        for(i=0;i<4;i++)        {
               printf("Name [%d] :%s %s",i+1,list[i].fname,list[i].lname);
               printf("\t\tPhone :%s\n",list[i].phone);
        }
        getch();
}
```

In above example the structure of array is declared

struct entry list[4];

This provides space in memory for 4 structure of type struct entry. The syntax we used to reference each element of the array list is similar to the syntax used for array of ints and chars. For example, we refer to zeroth list's fname as list [0].fname. Similarly, we refer first list's phone number as list [1].phone.

ARRAY WITHIN STRUCTURE:

```
#include <stdio.h>
#include <stddef.h>
int main(void) {
   #define SIZE 2
   struct test    {
      long type;
      size_t files[SIZE];
    };
   struct test example;
   example.type = 100;
   example.files[0] = 2;
   printf("example.type = %ld, example.files[0] = %lu\n",
        example.type, (unsigned long)example.files[0]);
   return 0;
}
output: example.type=100 , example.files[0]=2
```

Pointer with Structure

```
#include <stdio.h>
#include <string.h>
 struct student {
        int id;
        char name[30];
        float percentage;
};
 void main()  {
    struct student s1 = {1, "ABC", 90.5};
    struct student *ptr;
    ptr = &s1;
    printf("\n  Records of  STUDENT-1  ");
    printf("\n  Id is: %d ", ptr->id);
    printf("\n  Name is: %s ", ptr->name);
    printf("\n  Percentage is: %f  ", ptr->percentage);
}
```

In this program, "s1" is normal structure variable and "ptr" is pointer structure variable. As you know, Dot (.) operator is used to access the data using normal structure variable and arrow (->) is used to access data using pointer variable.

Passing structure to function

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address (reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

PASSING STRUCTURE TO FUNCTION IN C:

It can be done in below 3 ways.

- Passing structure to a function by value
- Passing structure to a function by address(reference)
- No need to pass a structure – Declare structure variable as global

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY VALUE:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

Program

```
#include <stdio.h>
#include <string.h>
void func(struct student record);
struct student {
        int id;
        char name[20];
        float percentage;
};
void main() {
        struct student record;
        record.id=1;
        strcpy(record.name, "ABC");
        record.percentage = 86.5;
        func(record);
 }
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: ABC
Percentage is: 86.500000
```

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

PROGRAM	OUTPUT

#include <stdio.h>
#include <string.h>
 void func(struct student *record);
 struct student {
        int id;
        char name[20];
        float percentage;
};
void main()
{
        struct student record;
         record.id=1;
        strcpy(record.name, " ABC ");
        record.percentage = 86.5;
         func(&record);
}
 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);
}
	Id is: 1
	Name is: ABC
	Percentage is: 86.500000

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:

Structure variables also can be declared as global variables as we declare other variables in C. So, when a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don't need to pass the structure to any function separately.

```
#include <stdio.h>
#include <string.h>
void structure_demo();
struct student {
        int id;
        char name[20];
        float percentage;
};
struct student record; // Global declaration of structure
void main()
{
        record.id=1;
        strcpy(record.name, " ABC ");
        record.percentage = 86.5;
         structure_demo();
}
```

```
 void structure_demo()
{
        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: ABC
Percentage is: 86.500000
```

Nested Structure in C

Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.

There are two ways to define nested structure in c language:

1. By separate structure
2. By Embedded structure

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.

```
#include <stdio.h>
#include <string.h>
 struct student_college_detail {
        int college_id;
         char college_name[50];
};
 struct student_detail  {
        int id;
        char name[20];
        float percentage;
        struct student_college_detail clg_data;  // structure within structure
}stu_data;
```

```
void main()
{
        struct student_detail stu_data = {1, "ABC", 90.5, 71145,"Saurashtra University"};
        printf(" Id is: %d \n", stu_data.id);
        printf(" Name is: %s \n", stu_data.name);
        printf(" Percentage is: %f \n\n", stu_data.percentage);
        printf(" College Id is: %d \n", stu_data.clg_data.college_id);
        printf(" College Name is: %s \n", stu_data.clg_data.college_name);
        getchar();
}
```

student_college_detail structure is declared inside **student_detail** structure in this program. Both structure variables are normal structure variables. Please note that members of **student_college_detail** structure are accessed by 2 dot(.) operator and members of **student_detail** structure are accessed by single dot(.) operator.

Embedded structure

We can define structure within the structure also. It requires less code than previous way. But it can't be used in many structures.

```
#include <stdio.h>
#include <string.h>
 struct student_college_detail
{
            int college_id;
             char college_name[50];
             struct student_detail
            {
                        int id;
                        char name[20];
                        float percentage;
            }stu_data;
};
void main()
{
struct student_college_detail clg_data = { 1,"Saurashtra University",{1, "ABC", 90.5}};
printf(" College Id is: %d \n", clg_data.college_id);
printf(" College Name is: %s \n",clg_data.college_name);
printf(" Id is: %d \n", clg_data.stu_data.id);
printf(" Name is: %s \n", clg_data.stu_data.name);
printf(" Percentage is: %f \n\n", clg_data.stu_data.percentage);
getchar();
}
```

UNION:

Union is a concept borrowed from structures and therefore follows the same syntax as structure. However there is major difference between them in terms of storage. In structures each member has its own storage location. Where as all the members of a union uses the same location. This implies that, although a union may contain many members of different types. It can handle only one member at a time. Like structures a union can be declared using the keyword union as follows:

```
union item
{
  int m;
  float x;
  char c;
} code;
```

This declares a variable code of type union item. The union contains three members each with a different data type. How ever, we can use only one of them at a time; this is due to the fact that only one location is allocated for a union variable.

<table>
<tr><td colspan="4">Storage of 4 bytes</td></tr>
<tr><td>1000</td><td>1001</td><td>1002</td><td>1003</td></tr>
<tr><td>c</td><td></td><td></td><td></td></tr>
<tr><td colspan="2">m</td><td></td><td></td></tr>
<tr><td colspan="4">x</td></tr>
</table>

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. This assumes that float variable requires 4 bytes of storage.
To access a union member, we can use the same syntax that we can use for structure members that is:

code.m;
code.x;
code.c;

all are valid member variable. During accessing we should make sure that we are accessing the member whose value is currently stored.

code.m=379;
code.x=7859.36;
printf("%d:",code.m);

A union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value the new value superseded the previous member's value.

STRUCTURE V/S UNION

BASIS OF COMPARISON	STRUCTURE	UNION
Basic	The separate memory location is allotted to each member of the 'structure'.	All members of the 'union' share the same memory location.

Declaration	**struct struct_name{
	type element1;
	type element2;
	.
	.
	} variable1, variable2, ...;**
		**union u_name{
		type element1;
		type element2;
		.
		.
		} variable1, variable2, ...;**

keyword	**'struct'**	**'union'**
Size	Size of Structure = sum of size of all the data members.	Size of Union = size of the largest members.
Store Value	Stores distinct values for all the members.	Stores same value for all the members.
At a Time	A 'structure' stores multiple values, of the different members, of the 'structure'.	A 'union' stores a single value at a time for all members.
Way of Viewing	Provide single way to view each memory location.	Provide multiple way to to view same memory location.
Anonymous feature	No Anonymous feature.	Anonymous union can be declared.
Access Members	We can access all the members of structure at anytime.	Only one member of union can be accessed at anytime.

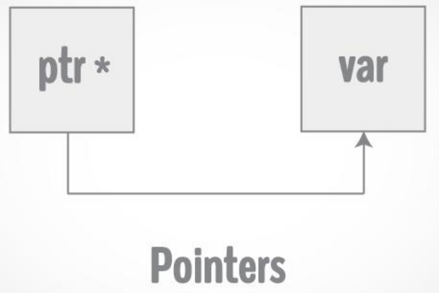
Example	**struct item_mst
	{
	int rno;
	char nm[50];
	}it;**
		**union item_mst
		{
		int rno;
		char nm[50];
		}it;**

UNIT 5
1. POINTERS IN C

POINTER INTRODUCTION:

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory



What are Pointers?

A pointer is a variable whose value is the address of another variable,

i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;          // pointer to an integer
double *dp;          // pointer to a double
float  *fp;          // pointer to a float
char   *ch           // pointer to a character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

NULL Pointers

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.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program

```
#include <stdio.h>
int main () {
  int  *ptr = NULL;
  printf("The value of ptr is : %x\n", ptr  );
  return 0;
}
```

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

The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing. To check for a null pointer, you can use an 'if' statement as follows

```
if(ptr)     // succeeds if p is not null
if(!ptr)    // succeeds if p is null
```

Pointer to Variable

There are a few important operations, which we will do with the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations

```
#include <stdio.h>
int main () {
  int  var = 20;          // actual variable declaration
  int  *ip;               // pointer variable declaration
  ip = &var;              // store address of var in pointer variable
  printf("Address of var variable: %x\n", &var  );    // address stored in pointer variable
  printf("Address stored in ip variable: %x\n", ip ); // access the value using the pointer
  printf("Value of *ip variable: %d\n", *ip );
  return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Reference operator (&) and Dereference operator (*)

As discussed, **& is called reference operator.** It gives you the address of a variable. Likewise, there is another operator that **gets you the value from the address, it is called a dereference operator (*)**. Above examples clearly demonstrate the use of pointers, reference operator and dereference operator.

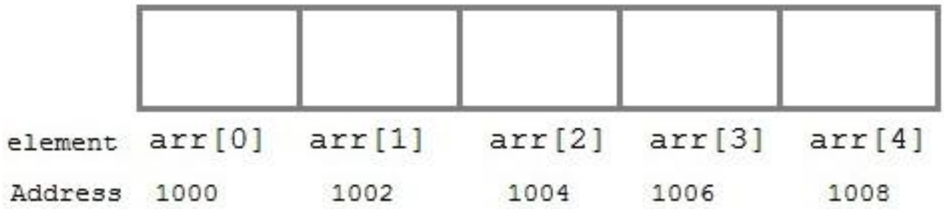
Note: The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Pointer to Array

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 elements will be stored as follows



Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000.

arr is equal to &arr[0]   // by default

We can declare a pointer of type int to point to the array arr.

int *p;
p = arr;
or p = &arr[0]; //both the statements are equivalent.

Now we can access every element of array arr using p++ to move from one element to another. As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array. Let's have an example,

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

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.

Pointer to Multidimensional Array

A multidimensional array is of form, a[i][j]. Let's see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], a will give the base address of this array, even a+0+0 will also give the base address, that is the address of a[0][0] element. Here is the generalized form for using pointer with multidimensional arrays *(*(ptr + i) + j) is same as a[i][j]

Pointer and Character strings

Pointer can also be used to create strings. Pointer variables of char type are treated as string.

char *str = "Hello";

This creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello". Another important thing to note that string created using char pointer can be assigned a value at runtime.

char *str;
str = "hello";   // this is Legal

The content of the string can be printed using printf() and puts().

printf("%s", str);

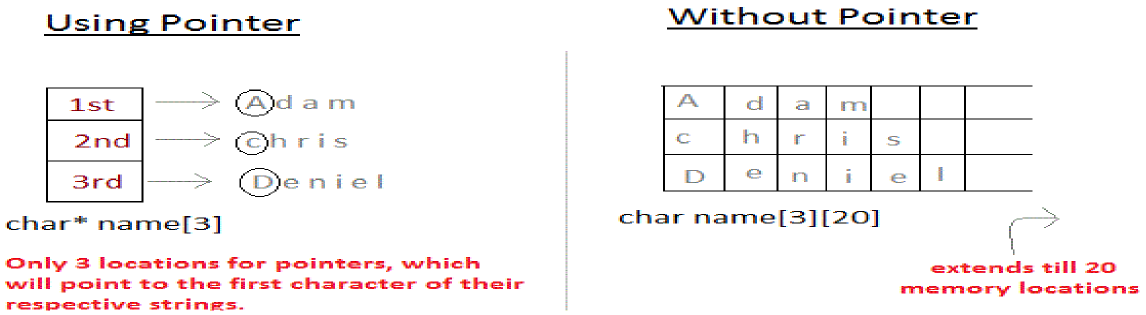
```
puts(str);
```

Notice that str is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator *.

Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

char *name[3]={
"Adam",
"chris",
"Deniel"
};
	//Now see same array without using pointer
	char name[3][20]= {
	"Adam",
	"chris",
	"Deniel"
	};


Using Pointer
1st
2nd
3rd
Adam
chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
Without Pointer
char name[3][20]
extends till 20 memory locations

A	d	a	m			
c	h	r	i	s		
D	e	n	i	e	l	

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

```
#include <stdio.h>
#include <conio.h>
 const int MAX = 3;
 void main () {
   char *names[] = {
                    "Adam",
                   "Chris",
                   "Deniel"
   };
  int i = 0;
  for ( i = 0; i < MAX; i++) {
          printf("Value of names[%d] = %s\n", i, names[i] );
   }
          getch();
}
```

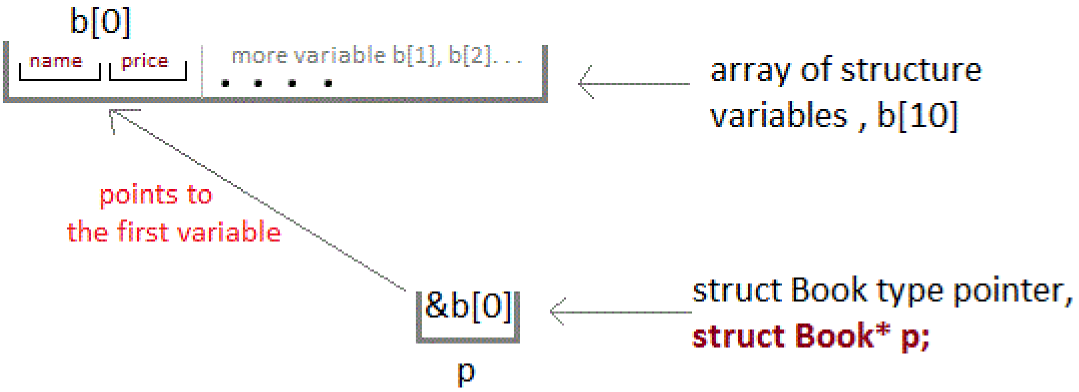
When the above code is compiled and executed, it produces the following result

```
Value of names[0] = Adam
Value of names[1] = Chris
Value of names[2] = Deniel
```

Pointer to Structure

We have array of integers, array of pointer etc, we can also have array of structure variables. And to make the use of array of structure variables efficient, we use pointers of structure type. We can also have pointer to a single structure variable, but it is mostly used with array of structure variables.

```
struct Book
{
        char name[10];
        int price;
};
int main()
{
        struct Book a;      //Single structure variable
        struct Book* ptr;   //Pointer of Structure type
        ptr = &a;
        struct Book b[10];   //Array of structure variables
        struct Book* p;      //Pointer of Structure type
        p = &b;
}
```


b[0]
name
price
more variable b[1], b[2]. . .
array of structure variables , b[10]
points to the first variable
&b[0]
p
struct Book type pointer,
struct Book* p;

Accessing Structure Members with Pointer

To access members of structure with structure variable, we used the dot. Operator. But when we have a pointer of structure type, we use arrow -> to access structure members.

```
struct Book
{
        char name[10];
        int price;
};
int main()
{
        struct Book b;
        struct Book* ptr = &b;
        ptr->name = "Dan Brown";     //Accessing Structure Members
        ptr->price = 500;
}
```

Pointer within Structure

```
struct Sample
{
	int  *ptr;  //Stores address of integer Variable
	char *name; //Stores address of Character String
}s1;
```

Structure may contain the Pointer variable as member. Pointers are used to store the address of memory location. They can be de-referenced by **"*"** operator. S1 is structure variable which is used to access the **"structure members"**.

```
		s1.ptr = &num;
		s1.name = "ABC"
```

Here num is any variable but it's address is stored in the Structure member ptr **(Pointer to Integer)** Similarly Starting address of the String "ABC" is stored in structure variable name **(Pointer to Character array)** Whenever we need to print the content of variable num , we are dereferencing the pointer variable num.

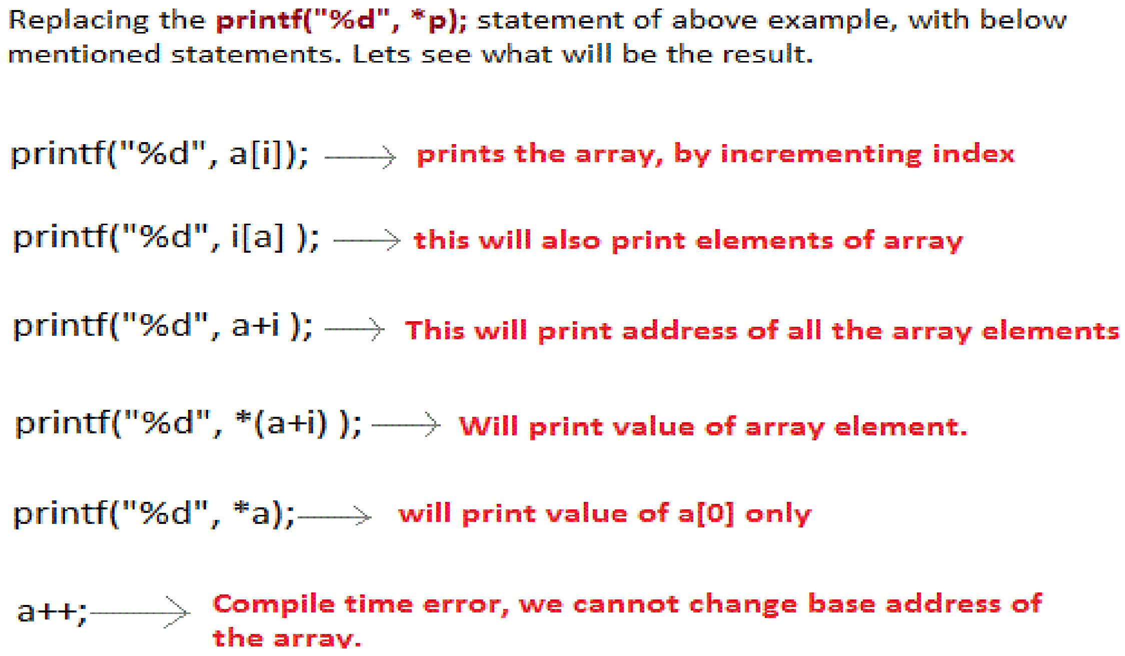
```
	printf("Content of Num : %d ",*s1.ptr);
	printf("Name : %s",s1.name);
	printf("\nRoll Number of Student : %d",*s1.ptr);
```

```
#include<stdio.h>
struct Student
{
	int  *ptr;			//Stores address of integer Variable
	char *name;		//Stores address of Character String
}s1;
void main()
{
	int roll = 20;
	s1.ptr   = &roll;
	s1.name  = "ABC";
	printf("\nRoll Number of Student : %d",*s1.ptr);
	printf("\nName of Student      : %s",s1.name);
	getch();
}
Output :
	Roll Number of Student : 20
	Name of Student      : ABC
```

We have stored the address of variable 'roll' in a pointer member of structure thus we can access value of pointer member directly using de-reference operator.

```
	printf("\nName of Student      : %s",s1.name);
```

Similarly we have stored the base address of string to pointer variable 'name'. In order to de-reference a string we never use de-reference operator.



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

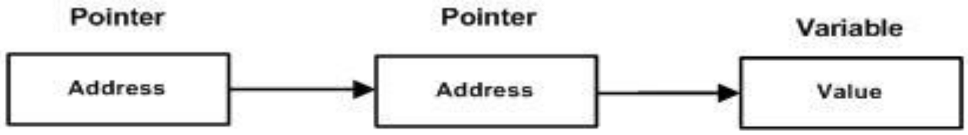
a++; → Compile time error, we cannot change base address of the array.

PASSING ADDRESS OF STRUCTURE VARIABLE TO A FUNCTION

```
struct book
 {
        char name[25];
        char author[25];
        int callno;
 };
main()
{
             struct book b1={ " let us c","ABC",101};
             display(&b1);
}
display(struct book * ptr)
{
              printf("\n%s %s %s",ptr->name,ptr->auther,ptr->callno);
}
```

POINTER TO POINTER

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type

int **var;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example

```
#include <stdio.h>
#include<conio.h>
 void main () {
        int  var;
        int  *ptr;
        int  **pptr;
        var = 3000;

  ptr = &var;            // take the address of var
  pptr = &ptr;           // take the address of ptr using address of operator &

  printf("Value of var = %d\n", var );      // take the value using pptr
  printf("Value available at *ptr = %d\n", *ptr );
  printf("Value available at **pptr = %d\n", **pptr);
 getch();
}
```

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

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

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.

UNIT 5

2. FILE HANDLING IN C

A data file is a computer file which stores data to be used by a computer application or system. It generally does not refer to files that contain instructions or code to be executed (typically called program files), or to files which define the operation or structure of an application or system (which include configuration files, etc.); but specifically to information used as input, or written as output by some other software program. This is especially helpful when debugging a program.

Most computer programs work with files. This is because files help in storing information permanently. Database programs create files of information. Compilers read source files and generate executable files. A file itself is an ordered collection of bytes stored on a storage device like tape, magnetic disk, optical disc etc. Data files are files that store data, for later use, pertaining to a specific application

What is a File?

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure.

Types of Files

When dealing with files, there are two types of files you should know about:

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:

- Creating a new file
- Opening an existing file
- Closing a file
- Reading from and writing information to a file
- Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

```
FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the library function in the "stdio.h" header file: **fopen().**

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode")
```

Example

```
fopen("c:\\newprogram.txt","w");
```

fopen("c:\\oldprogram.bin","rb");

Let's suppose the file newprogram.txt doesn't exist in the location c drive. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.

The writing mode allows you to create and edit (overwrite) the contents of the file. Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'. The reading mode only allows you to read the file, you cannot write into the file.

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

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.

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

Writing to a text file : Write to a text file using fprintf()

```
#include <stdio.h>
void 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);
}
```

This program takes a number from user and stores in the file program.txt. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered. Reading from a text file.

Read from a text file using fscanf()

```
#include <stdio.h>
void 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);
        getch();
}
```

This program reads the integer present in the program.txt file and prints it onto the screen. If you successfully created the file from Example 1, running this program will get you the integer you entered. Other functions like fgetchar(), fputc() etc. can be used in similar way.

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the function fwrite(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

fwrite(address_data,size_data,numbers_data,pointer_to_file);

Writing to a binary file using fwrite()

```
#include <stdio.h>
struct threeNum {
  int n1, n2, n3;
};
void main() {
        int n;
        struct threeNum num;
        FILE *fptr;
        if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
                printf("Error! opening file"); // Program exits if the file pointer returns
        NULL.
                exit(1);
        }
        for(n = 1; n < 5; ++n)
        {
            num.n1 = n;
            num.n2 = 5n;
            num.n3 = 5n + 1;
            fwrite(&num, sizeof(struct threeNum), 1, fptr);
        }
        fclose(fptr);
        getch();
}
```

In this program, you create a new file program.bin in the C drive. We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num. Now, inside the for loop, we store the value into the file using fwrite. The first parameter takes the address of num and the second parameter takes the size of the structure threeNum. Since, we're only inserting one instance of num, the third parameter is 1. And, the last parameter *fptr points to the file we're storing the data. Finally, we close

Reading from a binary file

Function fread() also take 4 arguments similar to fwrite() function as above.

fread(address_data,size_data,numbers_data,pointer_to_file);

Reading from a binary file using fread()

```
#include <stdio.h>
struct threeNum {
  int n1, n2, n3;
};
void main() {
        int n;
        struct threeNum num;
        FILE *fptr;
        if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
                printf("Error! opening file");    // Program exits if the file pointer
returns NULL.
                exit(1);
        for(n = 1; n < 5; ++n)
          {
                  fread(&num, sizeof(struct threeNum), 1, fptr);
                  printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
          }
        fclose(fptr);
        getch();
}
```

In this program, you read the same file program.bin and loop through the records one by one. In simple terms, you read one threeNum record of threeNum size from the file pointed by ***fptr** into the structure num. You'll get the same records you inserted in above Example

putw() and getw() Function

putw(), getw() functions are file handling function in C programming language which is used to write an integer value into a file (putw) and read integer value from a file (getw). Please find below the description and syntax for above file handling functions.

File operation	Declaration & Description
putw()	**Declaration**: int putw(int number, FILE *fp); putw function is used to write an integer into a file. In a C program, we can write integer value in a file as below. putw(i, fp); where i – integer value fp – file pointer

getw()	**Declaration:** int getw(FILE *fp);
	getw function reads an integer value from a file pointed by fp. In a C program, we can read integer value from a file as below.
	getw(fp);

```
#include<stdio.h>
#include<conio.h>
void main()  {
            FILE *fp;
            int n;
            clrscr();
            fp=fopen("c.dat", "wb+");
            printf("Enter the integer data");
            scanf("%d",&n);
            while(n!=0)       {
                        putw(n,fp);
                        scanf("%d",&n);
            }
rewind(fp);
          printf("Reading data from file");
          while((n=getw(fp))!=EOF)          {
                    printf("%d\n",n);
          }
          fclose(fp);
          getch();
}
```

fseek() Function

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record. This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek(). As the name suggests, fseek() seeks the cursor to the given record in the file.

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.

Whence	Value	Meaning
SEEK_SET	0	Starts the offset from the beginning of the file.
SEEK_END	1	Starts the offset from the end of the file.
SEEK_CUR	2	Starts the offset from the current location of the cursor in the file.

```
#include <stdio.h>
struct threeNum {
  int n1, n2, n3;
};
void main() {
          int n;
          struct threeNum num;
          FILE *fptr;
          if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
          printf("Error! opening file"); // Program exits if the file pointer returns
          NULL.
              exit(1);
          }
            fseek(fptr, sizeof(struct threeNum), SEEK_END);
           // Moves the cursor to the end of the file
          for(n = 1; n < 5; ++n) {
                    fread(&num, sizeof(struct threeNum), 1, fptr);
                    printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
          }
          fclose(fptr);
          getch();
}
```

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

rewind() Function

The rewind() function is used to move the cursor at the beginning of the file.

syntax:

void rewind(FILE *stream);

rewind() takes a file pointer and resets the position to the start of the file. For example the statement:

rewind(fp);

Would assign to fp because the file position has been set to the start of the file by rewind. This function helps us in reading a file more than once. Without having to close and open the file.

ftell() function

This function returns the value of the current pointer position in the file. The value is count from the beginning of the file.

Syntax:

long ftell(fptr);

Where fptr is a file pointer and function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value

```
#include <stdio.h>
void main () {
        FILE *fp;
        int len;
        fp = fopen("file.txt", "r");
        if( fp == NULL )  {
                perror ("Error opening file");
                return(-1);
        }
        fseek(fp, 0, SEEK_END);
        len = ftell(fp);
        fclose(fp);
        printf("Total size of file.txt = %d bytes\n",
len);
        getch();
}
```

freopen() Function

The freopen() associates a new filename with the given open stream and at the same time closes the old file in the stream.

Syntax:

FILE *freopen(const char *filename, const char *mode, FILE *stream)

Parameters

filename − This is the C string containing the name of the file to be opened.

mode − This is the C string containing a file access mode. It includes

Mode	Description
"r"	Opens a file for reading. The file must exist.
"w"	Creates an empty file for writing. If a file with the same name already exists then its content is erased and the file is considered as a new empty file.
"a"	Appends to a file. Writing operations appends data at the end of the file. The file is created if it does not exist.
"r+"	Opens a file to update both reading and writing. The file must exist.
"w+"	Creates an empty file for both reading and writing.
"a+"	Opens a file for reading and appending.

Stream – This is the pointer to a FILE object that identifies the stream to be re-opened. If the file was re-opened successfully, the function returns a pointer to an object identifying the stream or else, null pointer is returned.

```
#include <stdio.h>
void main () {
        FILE *fp;
        printf("This text is redirected to stdout\n");
        fp = freopen("file.txt", "w+", stdout);
        printf("This text is redirected to file.txt\n");
        fclose(fp);
        getch();
}
```

Let us compile and run the above program that will send the following line at STDOUT because initially we did not open stdout

This text is redirected to stdout

After a call to freopen(), it associates STDOUT to file file.txt, so whatever we write at STDOUT that goes inside file.txt. So, the file file.txt will have the following content.

This text is redirected to file.txt

remove() Function

The remove() function deletes the given filename so that it is no longer accessible.

Syntax:

int remove(const char *filename)

Parameters

filename – This is the C string containing the name of the file to be deleted. On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

```
#include <stdio.h>
#include <string.h>
void main () {
        int ret;
        FILE *fp;
        char filename[] = "file.txt";
        fp = fopen(filename, "w");
        fprintf(fp, "%s", "This is File Handling Example");
        fclose(fp);
        ret = remove(filename);
        if(ret == 0)   {
                printf("File deleted successfully");
        }
        else {
                   printf("Error: unable to delete the file");
        }
        getch();
}
```

Let us assume we have a text file file.txt having some content. So we are going to delete this file, above program. Let us compile and run the above program to produce the following message and the file will be deleted permanently.

File deleted successfully

rename() Function

The rename() changes the name of a file. You must close the file before renaming, as a file cannot be renamed if it is open.

Syntax:

int rename(const char *old_filename , const char *new_filename)

Parameters

old_filename – This is the C string containing the name of the file to be renamed and/or moved.

new_filename – This is the C string containing the new name for the file.

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

```
#include <stdio.h>
void main () {
        int ret;
        char oldname[] = "file.txt";
        char newname[] = "newfile.txt";
        ret = rename(oldname, newname);
        if(ret == 0)  {
                printf("File renamed
        successfully");
        }
        else    {
                printf("Error: unable to rename
        the file");
        }
        getch();
}
```

Let us assume we have a text file file.txt, having some content. So, we are going to rename this file, above program. Let us compile and run the above program to produce the following message and the file will be renamed to newfile.txt file.

feof() function (Test for End-of-File)

The feof() function tests to see if the end-of-file indicator has been set for a stream pointed to by stream.

Syntax:

int feof(FILE *stream)

Parameters

stream – This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.

```
#include <stdio.h>
void main () {
        FILE *fp;
        int c;
        fp = fopen("file.txt","r");
        if(fp == NULL)    {
                perror("Error in opening file");
                return(-1);
        }
        while(1)        {
        c = fgetc(fp);
        if( feof(fp) ) {
                break ;
        }
        printf("%c", c);
  }
        fclose(fp);
        getch();
}
```

ferror() Function

The ferror() function int ferror(FILE *stream) tests the error indicator for the given stream.

Syntax:

int ferror(FILE *stream)

Parameters

stream − This is the pointer to a FILE object that identifies the stream.

Return Value

If the error indicator associated with the stream was set, the function returns a non-zero value else, it returns a zero value.

```
#include <stdio.h>
void main()
{
        FILE *fp;
        char c;
        fp = fopen("file.txt", "w");
        c = fgetc(fp);
        if( ferror(fp) ) {
                    printf("Error in reading from file : file.txt\n");
        }
        clearerr(fp);
        if( ferror(fp)) {
                 printf("Error in reading from file : file.txt\n");
        }
        fclose(fp);
}
```

Assuming we have a text file file.txt, which is an empty file. Let us compile and run the above program that will produce the following result because we try to read a file which we opened in write only mode. Error reading from file "file.txt"

fflush() Function

The fflush() flushes the output/input buffer of a stream.

Syntax:

int fflush(FILE *stream)

Parameters

stream − This is the pointer to a FILE object that specifies a buffered stream.

Return Value

This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set.

```
#include <stdio.h>
#include <string.h>
void main() {
        char buff[1024];
        memset( buff, '\0', sizeof( buff ));
        fprintf(stdout, "Going to set full buffering on\n");
        setvbuf(stdout, buff, _IOFBF, 1024);
        fprintf(stdout, "This is tutorialspoint.com\n");
        fprintf(stdout, "This output will go into buff\n");
        fflush( stdout );
        fprintf(stdout, "and this will appear when programm\n");
        fprintf(stdout, "will come after sleeping 5 seconds\n");
        sleep(5);
 }
```

Let us compile and run the above program that will produce the following result. Here program keeps buffering into the output into buff until it faces first call to fflush(), after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before program comes out.

O/P

```
Going to set full buffering on
This is tutorialspoint.com
This output will go into buff
and this will appear when programm
will come after sleeping 5 seconds
```

fgetpos() Function

The fgetpos() function gets the current file position of the stream and writes it to pos.

Syntax:

int fgetpos(FILE *stream, fpos_t *pos)

Parameters

stream − This is the pointer to a FILE object that identifies the stream.

pos − This is the pointer to a fpos_t object.

Return Value

This function returns zero on success, else non-zero value in case of an error.

```
#include <stdio.h>
voiid main () {
        FILE *fp;
        fpos_t position;
        fp = fopen("file.txt","w+");
        fgetpos(fp, &position);
        fputs("Hello, World!", fp);
        fsetpos(fp, &position);
        fputs("This is going to override
previous content", fp);
        fclose(fp);
        getch();
}
```

Let us compile and run the above program to create a file file.txt which will have the following content. First of all we get the initial position of the file using fgetpos() function and then we write Hello, World! in the file, but later we have used fsetpos() function to reset the write pointer at the beginning of the file and then over-write the file with the following content:

O/P

```
This is going to override previous content
```

sprintf() Function

The sprintf()function sends formatted output to a string pointed to, by str.

Syntax:

int sprintf(char *str, const char *format, ...)

Parameters

str − This is the pointer to an array of char elements where the resulting C string is stored.

Format − This is the String that contains the text to be written to buffer. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags

prototype: %[flags][width][.precision][length]specifier,

explained below

Specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f.
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

Flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers − this is the number of digits to be printed after the decimal point. For g and G specifiers − This is the maximum number of significant digits to be printed. For s − this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type − it has no effect. When no precision is specified, the default is 1. If the period is

	specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Length	Description
H	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
L	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers − e, E, f, g and G).

Additional arguments − Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each % - tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned excluding the null-character appended at the end of the string, otherwise a negative number is returned in case of failure.

```
#include <stdio.h>
#include <math.h>
void main() {
        char str[80];
        sprintf(str, "Value of Pi = %f", M_PI);
        puts(str);
        getch();
}
```

O/P

```
Value of Pi = 3.141593
```

fscanf() Function

The fscanf() function reads formatted input from a stream.

Syntex:

int fscanf(FILE *stream, const char *format, ...)

Parameters

stream − This is the pointer to a FILE object that identifies the stream.

Format

This is the C string that contains one or more of the following items − Whitespace character, Non-whitespace character and Format specifiers. A format specifier will be as [=%[*][width][modifiers]type=], which is explained below −

Argument	Description
*	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
width	This specifies the maximum number of characters to be read in the current reading operation.
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the

	case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers

Type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally proceeded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

additional arguments

Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

```
#include <stdio.h>
#include <stdlib.h>
void main (){
        char str1[10], str2[10], str3[10];
        int year;
        FILE * fp;
        fp = fopen ("file.txt", "w+");
        fputs("We are in 2017", fp);
        rewind(fp);
        fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
        printf("Read String1 |%s|\n", str1 );
        printf("Read String2 |%s|\n", str2 );
        printf("Read String3 |%s|\n", str3 );
        printf("Read Integer |%d|\n", year );
        fclose(fp);
        getch();
}
O/P
        Read 0String1 |We|
        Read String2 |are|
        Read String3 |in|
        Read Integer |2017|
```

vsprintf() function

The vsprintf() function sends formatted output to a string using an argument list passed to it.

Syntax:

int vsprintf(char *str, const char *format, va_list arg)

Parameters

str − This is the array of char elements where the resulting string is to be stored.

format − This is the C string that contains the text to be written to the str. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and are formatted as requested.

Format tags prototype %[flags][width][.precision][length]specifier, as explained below

Specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f.
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

Flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with

	e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers − this is the number of digits to be printed after the decimal point. For g and G specifiers − This is the maximum number of significant digits to be printed. For s − this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type − it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers − i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers − e, E, f, g and G).

arg − An object representing the variable arguments list. This should be initialized by the va_start macro defined in **<stdarg>**.

Return Value

If successful, the total number of characters written is returned, otherwise a negative number is returned.

```
#include <stdio.h>
#include <stdarg.h>
char buffer[80];
int vspfunc(char *format, ...) {
        va_list aptr;
        int ret;
        va_start(aptr, format);
        ret = vsprintf(buffer, format, aptr);
        va_end(aptr);
        return(ret);
        }
```

```
void  main() {
        int i = 5;
        float f = 27.0;
        char str[50] = "Smt. J.J.K.C.C";
        vspfunc("%d %f %s", i, f, str);
        printf("%s\n", buffer);
        getch();
  }
```

Let us compile and run the above program, this will produce the following result −

O/P

```
        5 27.000000 Smt. J.J.K.C.C
```

setbuf() function

The setbuf() function defines how a stream should be buffered. This function should be called once the file associated with the stream has already been opened, but before any input or output operation has taken place.

Syntax

void setbuf(FILE *stream, char *buffer)

Parameters

stream − This is the pointer to a FILE object that identifies an open stream.

buffer − This is the user allocated buffer. This should have a length of at least BUFSIZ bytes, which is a macro constant to be used as the length of this array.

Return Value

This function does not return any value.

```
#include <stdio.h>
void main() {
        char buf[BUFSIZ];
        setbuf(stdout, buf);
        puts("Smt. J. J. K. C. C.");
        fflush(stdout);
        getch();
}
```

Let us compile and run the above program to produce the following result. Here program sends output to the STDOUT just before it comes out, otherwise it keeps buffering the output. You can also use fflush() function to flush the output.

O/P

```
        Smt. J. J. K. C. C.
```

setvbuf() function

The setvbuf() function defines how a stream should be buffered.

syntax

int setvbuf(FILE *stream, char *buffer, int mode, size_t size)

Parameters

stream − This is the pointer to a FILE object that identifies an open stream.

buffer − This is the user allocated buffer. If set to NULL, the function automatically allocates a buffer of the specified size.

mode − This specifies a mode for file buffering −

Mode	Description
_IOFBF	Full buffering − On output, data is written once the buffer is full. On Input the buffer is filled when an input operation is requested and the buffer is empty.
_IOLBF	Line buffering − On output, data is written when a newline character is inserted into the stream or when the buffer is full, what so ever happens first. On Input, the buffer is filled till the next newline character when an input operation is requested and buffer is empty.
_IONBF	No buffering − No buffer is used. Each I/O operation is written as soon as possible. The buffer and size parameters are ignored.

size − This is the buffer size in bytes

Return Value

This function returns zero on success else, non-zero value is returned.

```
#include <stdio.h>
void main(){
	char buff[1024];
	memset( buff, '\0', sizeof( buff ));
	fprintf(stdout, "Going to set full buffering on\n");
	setvbuf(stdout, buff, _IOFBF, 1024);
	fprintf(stdout, "This output will go into buff\n");
	fflush( stdout );
	fprintf(stdout, "and this will appear when program\n");
	fprintf(stdout, "will come after sleeping 5 seconds\n");
	sleep(5);
	getch();
}
```

Let us compile and run the above program to produce the following result. Here program keeps buffering the output into buff until it faces first call to fflush(), after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before the program comes out.

O/P

```
	Going to set full buffering on
	This output will go into buff
	and this will appear when program
	will come after sleeping 5 seconds
```

vsnprintf() function

Write formatted data from variable argument list to sized buffer Composes a string with the same text that would be printed if format was used on printf, but using the elements in the variable argument list identified by arg instead of additional function arguments and storing the resulting content as a C string in the buffer pointed by s (taking n as the maximum buffer capacity to fill).

If the resulting string would be longer than n-1 characters, the remaining characters are discarded and not stored, but counted for the value returned by the function.

Internally, the function retrieves arguments from the list identified by arg as if va_arg was used on it, and thus the state of arg is likely to be altered by the call.

In any case, arg should have been initialized by va_start at some point before the call, and it is expected to be released by va_end at some point after the call.

Syntax

```
		int vsnprintf (char * s, size_t n, const char * format, va_list arg );
```

Parameters

Mode	Description
s	Pointer to a buffer where the resulting C-string is stored. The buffer should have a size of at least n characters.
n	Maximum number of bytes to be used in the buffer. The generated string has a length of at most n-1, leaving space for the additional terminating null character. size_t is an unsigned integral type
format	C string that contains a format string that follows the same specifications as format in printf (see printf for details).
arg	A value identifying a variable arguments list initialized with va_start. va_list is a special type defined in <cstdarg>.

Return Value

The number of characters that would have been written if n had been sufficiently large, not counting the terminating null character. If an encoding error occurs, a negative number is returned. Notice that only when this returned value is non-negative and less than n, the string has been completely written.

In this example, if the file myfile.txt does not exist, perror is called to show an error message similar to:

```
#include <stdio.h>
#include <stdarg.h>
void PrintFError ( const char * format, ... ) {
            char buffer[256];
             va_list args;
            va_start (args, format);
            vsnprintf (buffer,256,format, args);
            perror (buffer);
            va_end (args);
}
void main () {
            FILE * pFile;
            char szFileName[]="myfile.txt";
            pFile = fopen (szFileName,"r");
            if (pFile == NULL)
                        PrintFError ("Error opening '%s'",szFileName);
            else  {
                            fclose (pFile);    // file successfully open
             }
            getch();
}
```

O/P

```
Error opening file 'myfile.txt': No such file or directory
```

Command Line Arguments

main () function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,

- argc
- argv[]

where,

argc - Number of arguments in the command line including program name

argv[] – This is carrying all the arguments

- In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.
- For example, when we compile a program (test.c), we get executable file in the name “test”.
- Now, we run the executable “test” along with 4 arguments in command line like below.

```
. /test this is a program
```

Where,

```
        argc        =     5
        argv[0]     =     “test”
        argv[1]     =     “this”
        argv[2]     =     “is”
        argv[3]     =     “a”
        argv[4]     =     “program”
        argv[5]     =     NULL
```

Example program for argc() and argv() functions in C:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])   //  command line arguments
{
          if(argc!=5)
          {
            printf("Arguments passed through command line not equal to 5");
            return 1;
}
          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]);
          return 0;
}
```

Output:

```
        Program name : test
        1st arg : this
        2nd arg : is
        3rd arg : a
        4th arg : program
        5th arg : (null)
```