



WEB DEVELOPMENT | DIGITAL MARKETING | CONSULTANCY

C - Programming

By Himanshu Tyagi
+91-9756042019

C – Programming (Introduction)

The C Language is developed for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc. C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

- Mother language.
- System programming language.
- Procedure-oriented programming language.
- Structured programming language.
- Mid-level programming language.

History of C Language :-

History of C language is interesting to know. Here we are going to discuss a brief history of the c language. **C programming language** was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

C- Features

C is the widely used language. It provides many **features** that are given below.

Features

- Simple
- Machine Independent or Portable
- Mid-level programming language
- structured programming language
- Rich Library
- Memory Management
- Fast Speed
- Pointers
- Recursion
- Extensible

Cons:

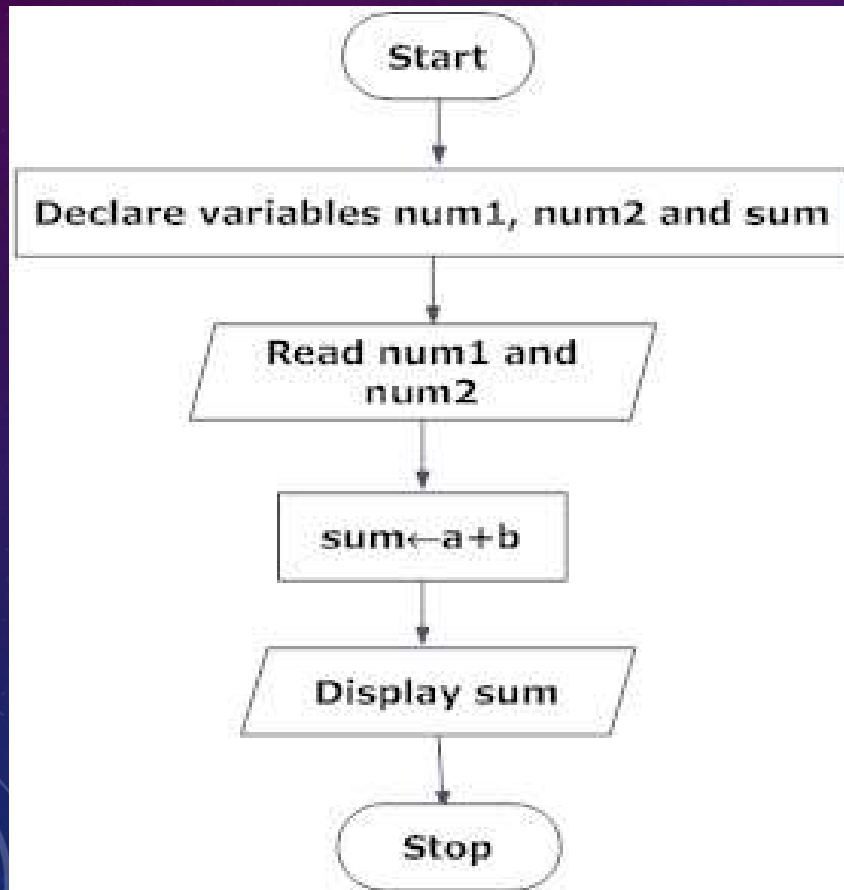
1. Data security
2. No run-time checking:
3. No strict type checking:
4. No code-reuse:
5. Namespace concept:
6. No OOP concepts:
7. Effects on Today's programming:
8. Real-world problems
9. Extending the program issues:
10. High-level constructs: .

Pros:

1. Portable language:
2. Building block for other languages:
3. Structured programming language:
4. Easy to learn.
5. Built-in function
6. User-defined function.
7. Explore hidden objects
8. Speed-up programs.
9. Compile
10. Low level of abstraction

C-Flow Chart

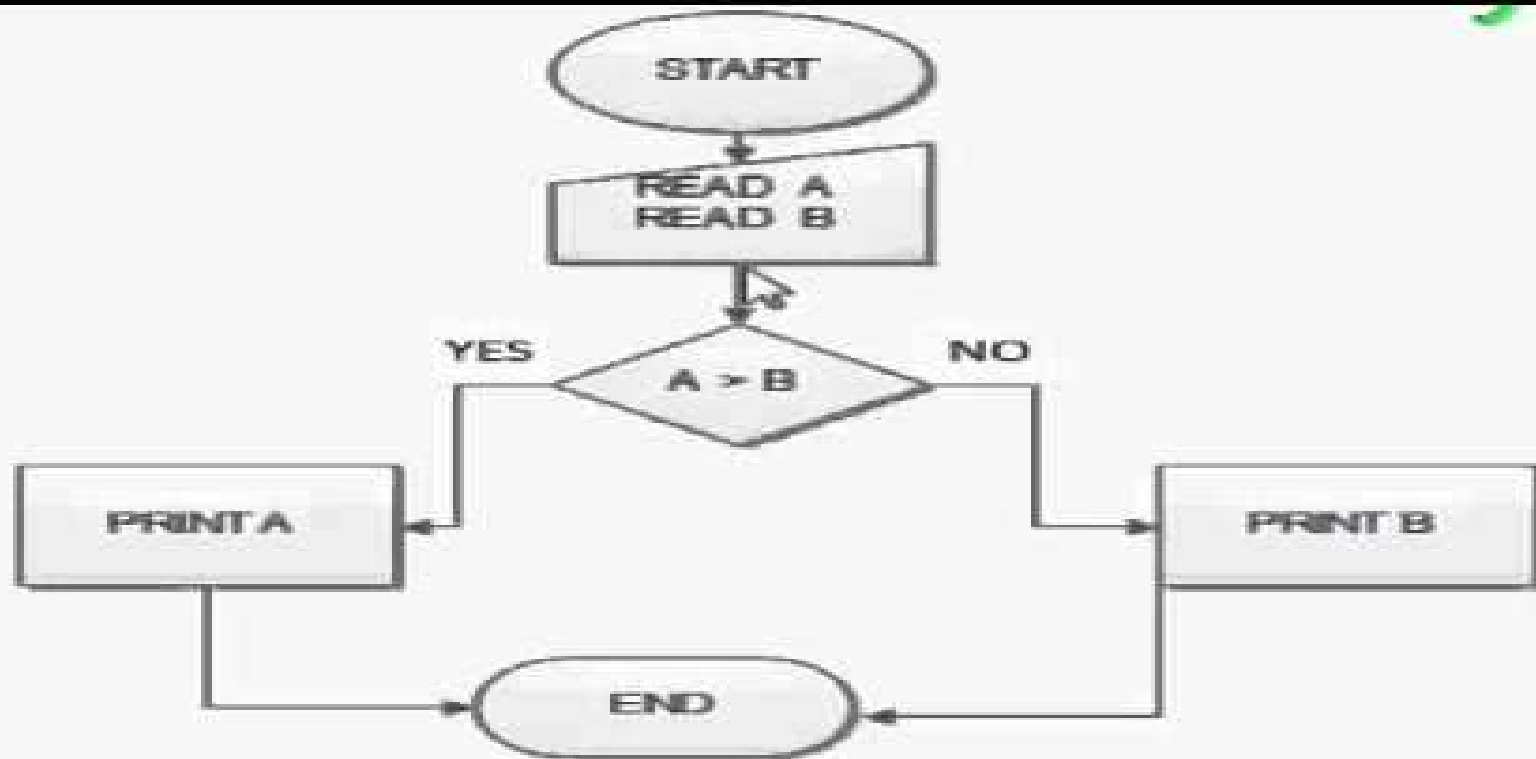
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing. The process of drawing a flowchart for an algorithm is known as “flowcharting”.



	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for airthmetic operations and data-manipulations.
	Desicion	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect flowchart portion on different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

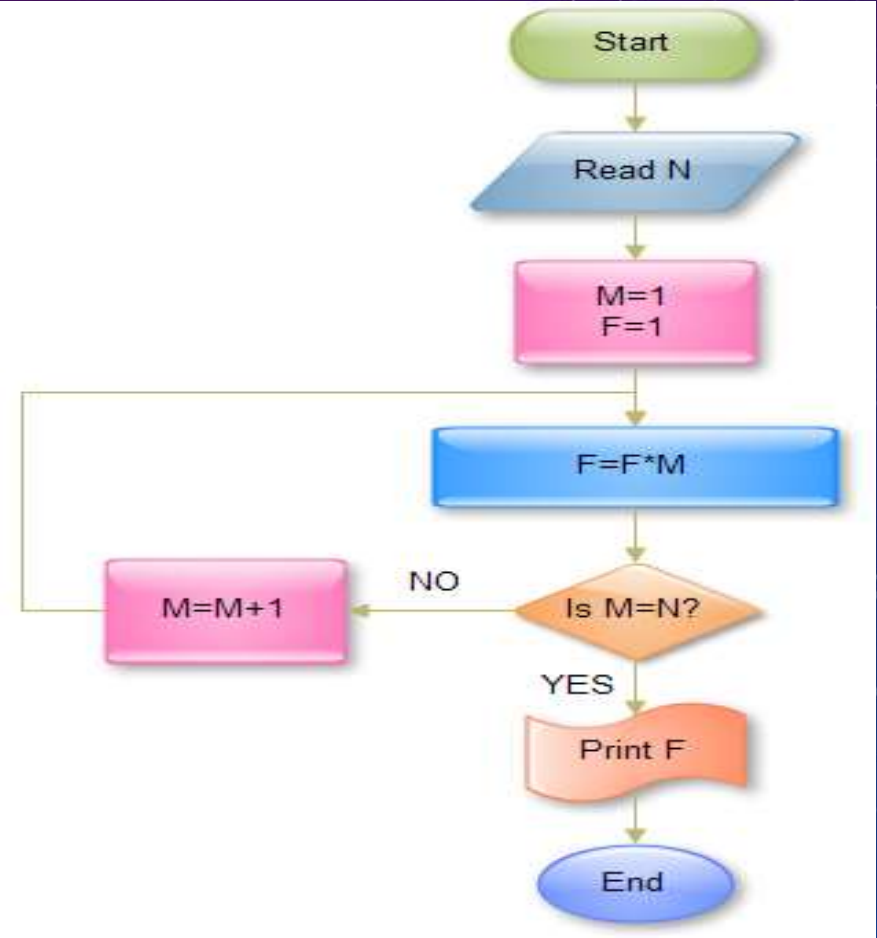
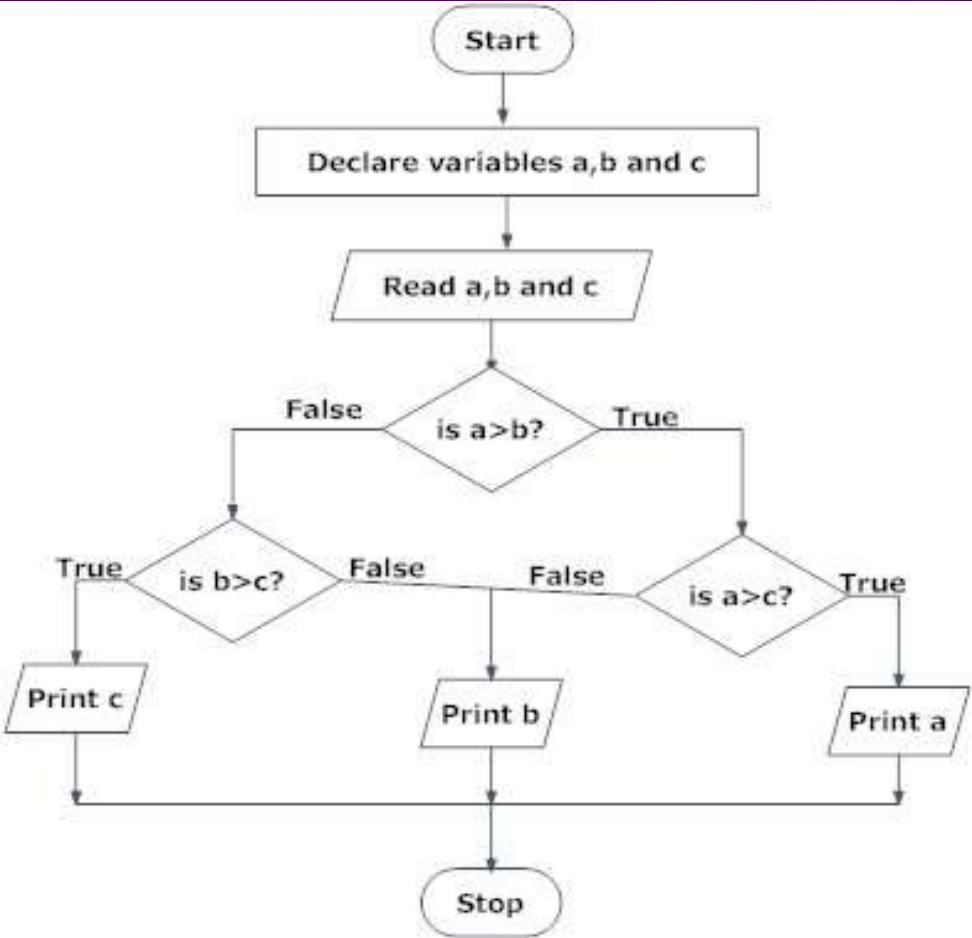
C- Flow Chart

Examples for understanding the concepts of Flow Chart:



C-Flow Chart

Example for Flow chart



C - Algorithm

Algorithm is a procedure or step-by-step instruction for solving a problem. They form the foundation of writing a program. Algorithm can be written as a list of steps using text or as a picture with shapes and arrows called a flowchart.

For writing any programs, the following has to be known:

- Input
- Tasks to be performed
- Output expected

Qualities of a good algorithm:

- Input and output should be defined precisely.
- Each steps in algorithm should be clear and unambiguous.
- Algorithm should be most effective among many different ways to solve a problem.
- An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Systematic logical approach which is a well-defined, step-by-step procedure that allows a computer to solve a problem.

Example : Write an algorithm to add two numbers entered by user.

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum. $\text{sum} \leftarrow \text{num1} + \text{num2}$

Step 5: Display sum Step 6: Stop

C-Algorithm

Example for finding greatest between three numbers.

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If $a > b$

 If $a > c$

 Display a is the largest number.

 Else

 Display c is the largest number.

Else

 If $b > c$

 Display b is the largest number.

 Else

 Display c is the greatest number.

Step 5: Stop

C - Algorithm

Example – Algorithm for Factorial program

Step 1: Start

Step 2: Declare variables n, factorial and i.

Step 3: Initialize variables

factorial \leftarrow 1

i \leftarrow 1

Step 4: Read value of n

Step 5: Repeat the steps until i=n

5.1: factorial \leftarrow factorial * i

5.2: i \leftarrow i + 1

Step 6: Display factorial

Step 7: Stop

C-Program, Algorithm and Pseudocode

Algorithm : Systematic logical approach which is a well-defined, step-by-step procedure that allows a computer to solve a problem.

Pseudocode : It is a **simpler version of a programming code** in plain English which uses short phrases to write code for a program before it is implemented in a specific programming language.

Program : It is exact code written for problem **following all the rules of the programming language.**

Algorithm of linear search :

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[].
2. If x matches with an element, return the index.
3. If x doesn't match with any of elements, return -1.

Pseudocode for Linear Search :

```
FUNCTION linearSearch(list, searchTerm):  
    FOR index FROM 0 -> length(list):  
        IF list[index] == searchTerm THEN  
            RETURN index  
        ENDIF  
    ENDLOOP  
    RETURN -1  
END FUNCTION
```

Program for Linear search

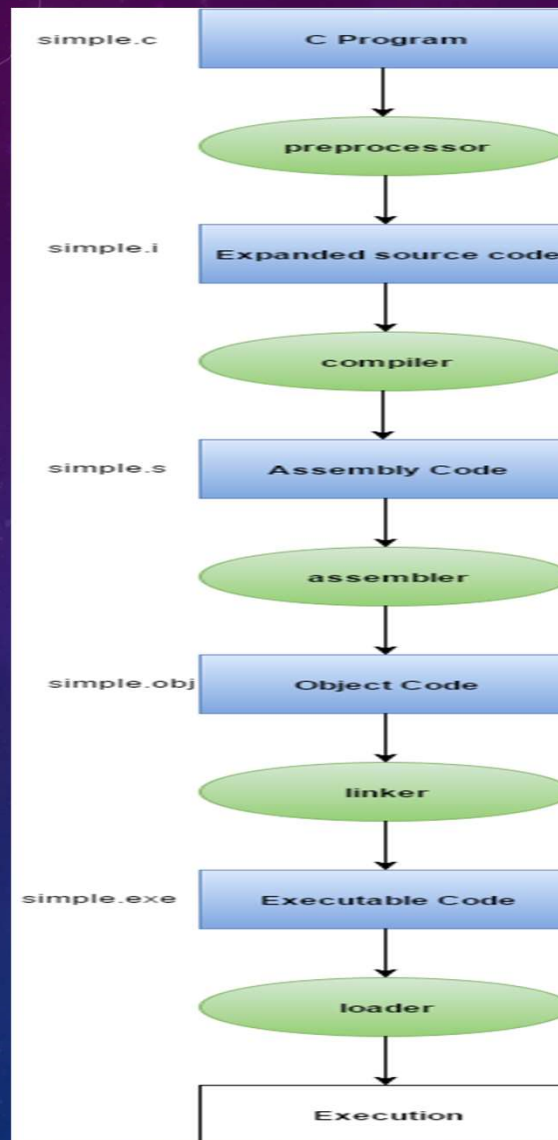
```
int search(int arr[], int n, int x)  
{  
    int i;  
    for (i = 0; i < n; i++)  
        if (arr[i] == x)  
            return i;  
    return -1;  
}
```

C-Program Explanation

```
#include <stdio.h>
int main()
{
    printf("Hello C Language");

    return 0;
}
```

- **#include <stdio.h>** includes the **standard input output** library functions.
- The **printf()** function is defined in **stdio.h**.
- **int main()** The **main()** function is the **entry point of every program** in c language.
- **printf()** The **printf()** function is **used to print data** on the console.
- **return 0** The **return 0** statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.



Let's try to understand the flow of above program by the figure given below.

- 1) C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. The preprocessor generates an expanded source code.
- 2) Expanded source code is sent to compiler which compiles the code and converts it into assembly code.
- 3) The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.
- 4) The object code is sent to linker which links it to the library such as header files. Then it is converted into executable code. A simple.exe file is generated.
- 5) The executable code is sent to loader which loads it into memory

C – Basic Syntax

Tokens:

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol.

For Example:- `printf("Hello, World! \n");`

- `Printf`
- `(`
- `“Hello,World!\n”`
- `)`
- `;`

Identifiers:

C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, \$, and % within identifiers. C is a **case-sensitive** programming language. Thus, *Man* and *man* are two different identifiers in C.

Semicolons:

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Comments:

Comments are like helping text in your C program and they are ignored by the compiler. They start with `/* Hello */`.

Basically comments is used to express the Source Code for Future use. And it also increase the readability of the source Code.

C- Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names. There are only 32 reserved words (keywords) in the C language.

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	typedef	union	unsigned	void	volatile	while

Whitespace :

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it. Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins.

printf() and scanf() :

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file). The **printf() function** is used for output. It prints the given statement to the console. The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

The **scanf() function** is used for input. It reads the input data from the console.

C – printf and scanf examples

E.g to print cube of a given number.

```
#include<stdio.h>
int main()
{
int n;
printf("enter a number ==:");
scanf("%d",&n);
printf("cube of number is:%d ",n*n*n);
return 0;
}
```

```
#include <stdio.h>
int main()
{
char ch;
char str[100];
printf("Enter any character \n");
scanf("%c", &ch);
printf("Entered character is %c \n", ch);
printf("Enter any string ( upto 100 character
) \n");
scanf("%s", &str);
printf("Entered string is %s \n", str);
```

E.g Sum of two numbers.

```
#include<stdio.h>
int main()
{
int x=0,y=0,result=0;
printf("enter first number:");
scanf("%d",&x);
printf("enter second number:");
scanf("%d",&y);
result=x+y;
printf("sum of 2 num:%d ",result);
return 0;
}
```

```
#include <stdio.h>
int main()
{
char ch = 'A';
char str[20] = "fresh2refresh.com";
float flt = 10.234;
int no = 150;
double dbl = 20.123456;
printf("Character is %c \n", ch);
printf("String is %s \n", str);
printf("Float value is %f \n", flt);
printf("Integer value is %d\n", no);
printf("Double value is %lf \n", dbl);
printf("Octal value is %o \n", no);
printf("Hexadecimal value is %x \n", no);
return 0;
}
```

- printf() is used to display the output and scanf() is used to read the inputs.
- printf() and scanf() functions are declared in “stdio.h” header file in C library.
- All syntax in C language including printf() and scanf() functions are case sensitive.

C-Variables

A variable is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times. It is a way to represent memory location through symbol so that it can be easily identified.

The example of declaring the variable is given below: `{ int a; float b; char c; }`

We can also provide values while declaring the variables : `int a=10,b=20; //declaring 2 variable of integer type`
`float f=20.8; , char c='A';`

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Types of Variables in C

- local variable - A variable that is declared inside the function or block is called a local variable.
- global variable - It is declared outside the function or block. Any function can change the value of it. It is available to all .
- static variable - variable that is declared with the static keyword is called static variable.
- automatic variable -variables in C that are declared , automatic var by default. We can explicitly declare it using **auto KW**.
- external variable - We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

lvalue – Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue – The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

C- Constant

Constants in C

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.
There are different types of constants in C programming.

Two ways to define constant in C

There are two ways to define constant in C programming.

- **const keyword** e.g. **const float** PI=3.14;
- **#define preprocessor** e.g.

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'
int main()
{
    int area;
    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);
    return 0;
}
```


C- Data Types

Data Types :A data type specifies the type of data that a variable can store such as integer, floating, character, etc.

Data Types	Memory Size	Range
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	−2,147,483,648 to 2,147,483,647
signed long int	4 byte	−2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

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

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

C – Escape Sequence

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character. It is composed of two or more characters starting with backslash \. For example: \n represents new line.

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

```
#include<stdio.h>
int main(){
    int number=50;
    printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\");
    return 0;
}
```

C - Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators .

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Ternary or Conditional Operators
- Assignment Operator
- Misc Operator

Precedence of Operators :

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

e.g. `int value=10+20*10;`

The value variable will contain **210** because * (multiplicative operator) is evaluated before + (additive operator).

C – Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

C - Operators

Arithmetic Operators:

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
-	Subtracts second operand from the first.	A - B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by denominator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

e.g,

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

```
main()
```

```
{ int a = 21; int b = 10; int c;
```

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

```
printf("Line 1 - Value of c is %d\n", c);
```

```
c = a - b;
```

```
printf("Line 2 - Value of c is %d\n", c);
```

```
c = a * b;
```

```
printf("Line 3 - Value of c is %d\n", c);
```

```
c = a / b;
```

```
printf("Line 4 - Value of c is %d\n", c);
```

```
c = a % b;
```

```
printf("Line 5 - Value of c is %d\n", c);
```

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

```
printf("Line 6 - Value of c is %d\n", c);
```

```
c = a--;
```

```
printf("Line 7 - Value of c is %d\n", c);
```

```
}
```

C - Operators

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 :

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

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

```
main() {
```

```
    int a = 21; int b = 10; int c ;
```

```
    if( a == b )
```

```
    {    printf("Line 1 - a is equal to b\n" ); }
```

```
    else {    printf("Line 1 - a is not equal to b\n" ); }
```

```
    if ( a < b ) {    printf("Line 2 - a is less than b\n" ); }
```

```
    else {    printf("Line 2 - a is not less than b\n" ); }
```

```
    if ( a > b ) {    printf("Line 3 - a is greater than b\n" ); }
```

```
    else {    printf("Line 3 - a is not greater than b\n" ); }
```

```
    /* Lets change value of a and b */
```

```
    a = 5; b = 20;
```

```
    if ( a <= b ) {    printf("Line 4 - a is either less than or equal to b\n" ); }
```

```
    if ( b >= a ) {    printf("Line 5 - b is either greater than or equal to a\n" ); }
```

```
}
```

C - Operators

Logical Operators

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

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

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


C - Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e., -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 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 = 15 i.e., 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 );}
```


C - Operators

Assignment Operators: The following table lists the assignment operators supported by the C language –

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

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

```
main()
```

```
{
```

```
    int a = 21;  int c ;
```

```
    c = a;  printf("Line 1 - = Operator Example, Value of c = %d\n", c );
```

```
    c += a;  printf("Line 2 - += Operator Example, Value of c = %d\n", c );
```

```
    c -= a;  printf("Line 3 - -= Operator Example, Value of c = %d\n", c );
```

```
    c *= a;  printf("Line 4 - *= Operator Example, Value of c = %d\n", c );
```

```
    c /= a;  printf("Line 5 - /= Operator Example, Value of c = %d\n", c );
```

```
    c = 200;
```

```
    c %= a;  printf("Line 6 - %= Operator Example, Value of c = %d\n", c );
```

```
    c <<= 2;  printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );
```

```
    c >>= 2;  printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );
```

```
    c &= 2;  printf("Line 9 - &= Operator Example, Value of c = %d\n", c );
```

```
    c ^= 2;  printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );
```

```
    c |= 2;  printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
```

```
}
```

C -Operators

Misc Operators \mapsto sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

```
#include <stdio.h>
main() {
    int a = 4;  short b;  double c;
    int* ptr; /* example of sizeof operator */
    printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
    printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
    printf("Line 3 - Size of variable c= %d\n", sizeof(c) ); /* example of &
and * operators */
    ptr = &a; /* 'ptr' now contains the address of 'a'*/
    printf("value of a is %d\n", a);
    printf("*ptr is %d.\n", *ptr); /* example of ternary operator */
    a = 10;  b = (a == 1) ? 20: 30;
    printf( "Value of b is %d\n", b );
    b = (a == 10) ? 20: 30;
    printf( "Value of b is %d\n", b );
}
```

C – Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. There are four different storage classes in a C program . Auto , register , static , extern

- AUTO Class - The **auto** storage class is the default storage class for all local variables. E.g `auto int a = 5;`
- REGISTER Class- The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location). E.g `= register int a = 5;`
The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.
- STATIC CLASS- The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- Extern Class- The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

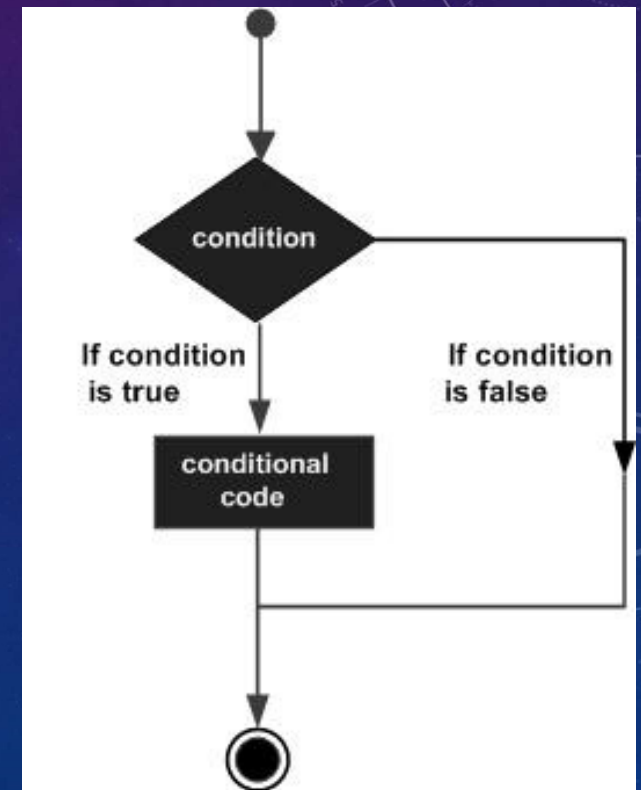
Control Structure--

Decision making structures require that the programmer specifies 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.

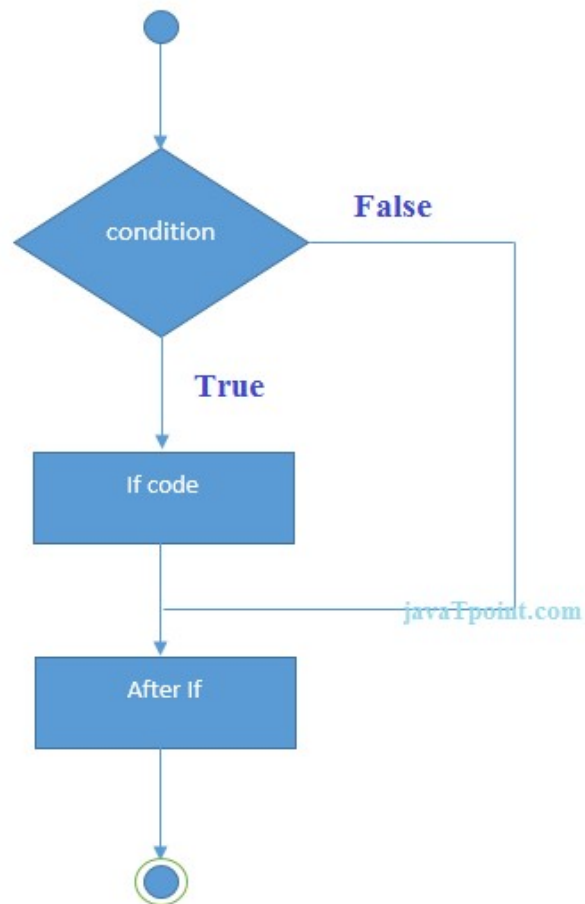
The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in C language.

- **If statement** --An **if statement** consists of a boolean expression followed by one or more statements.
- **If-else statement** -- An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
- **Nested if** -- You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
- **Switch statement** - A **switch** statement allows a variable to be tested for equality against a list of values.



If -statement

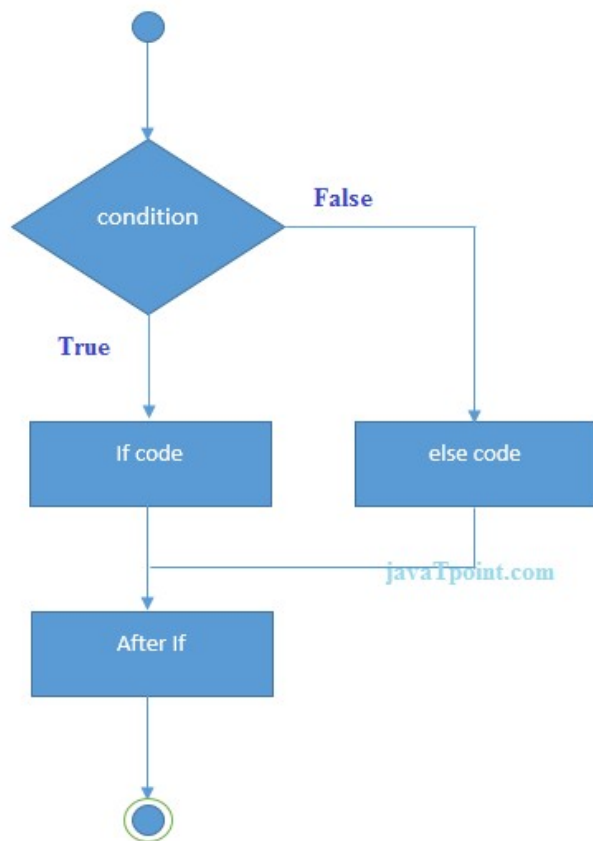


Find Number is Even or Odd

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

If-else statement

The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. We must notice that if and else block cannot be executed simultaneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition.

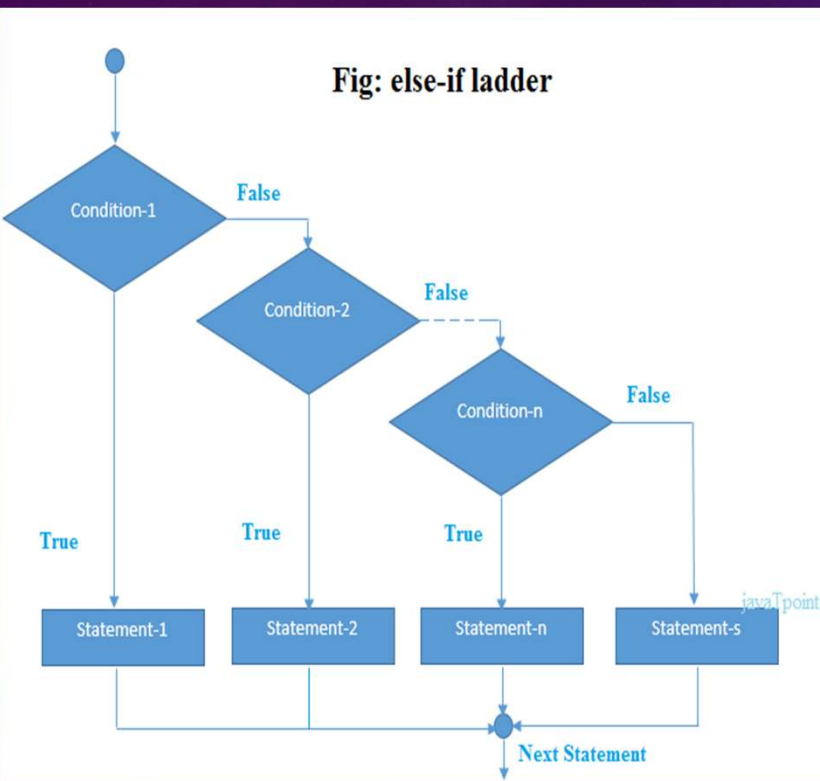


Ex-The given number is Even or odd

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

If else-if ladder Statement

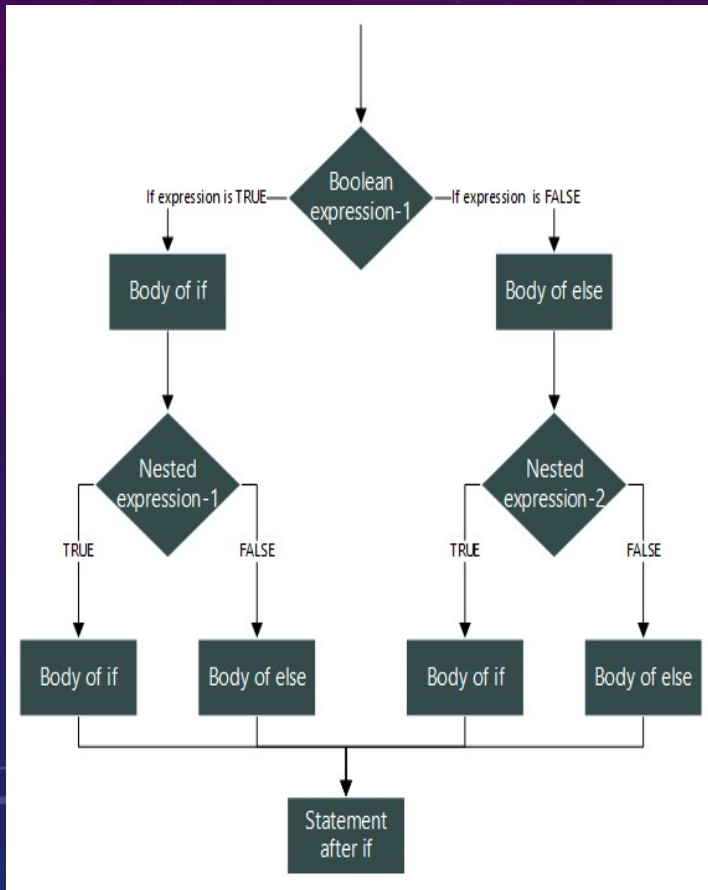
The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed.



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

Nested if else

It is always legal in C programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).



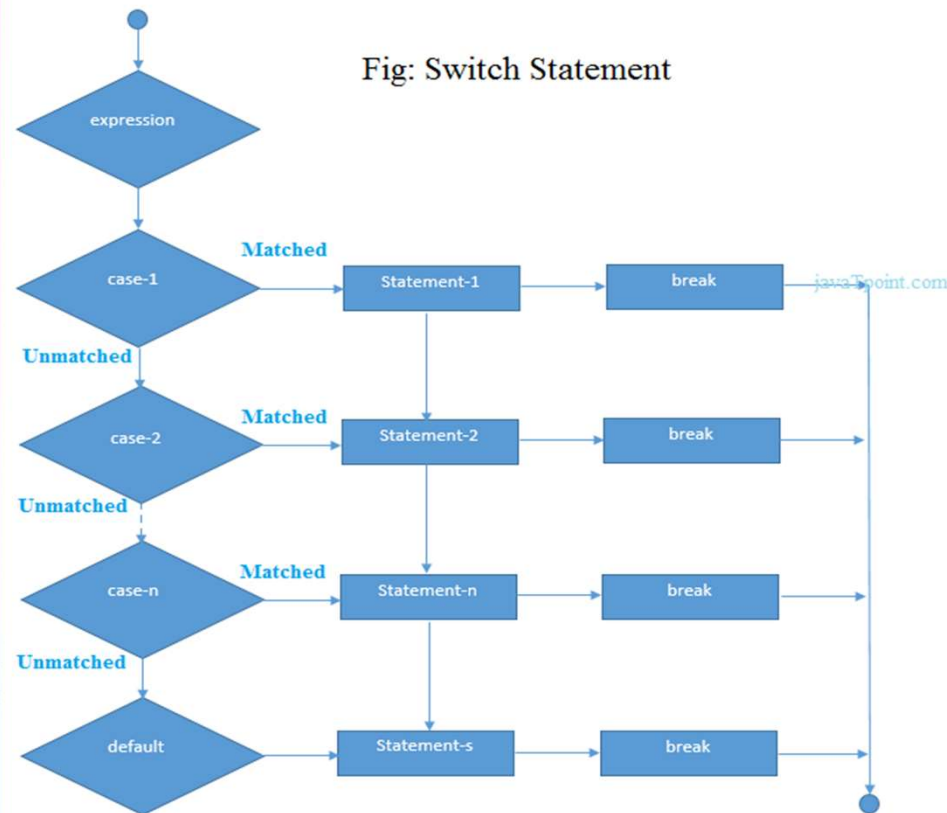
```
#include <stdio.h>
int main ()
{ /* local variable definition */
  int a = 100;  int b = 200;
  /* check the boolean condition */
  if( a == 100 ) {
    /* if condition is true then check the following */
    if( b == 200 ) {
      /* if condition is true then print the following */
      printf("Value of a is 100 and b is 200\n" );
    }
  }
  printf("Exact value of a is : %d\n", a );
  printf("Exact value of b is : %d\n", b );
  return 0;
}
```


Switch Case--- statement

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, We can define various statements in the multiple cases for the different values of a single variable.

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

Fig: Switch Statement



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

```
int main(){
```

```
int number=0;
```

```
printf("enter a number:");
```

```
scanf("%d",&number);
```

```
switch(number){
```

```
case 10:
```

```
printf("number is equals to 10");
```

```
break;
```

```
case 50:
```

```
printf("number is equal to 50");
```

```
break;
```

```
case 100:
```

```
printf("number is equal to 100");
```

```
break;
```

```
default:
```

```
printf("number is not equal to 10, 50 or 100");
```

```
}
```

```
return 0;
```

Switch Case – Rule based

Rules for switch statement in C language

- 1) The *switch expression* must be of an integer or character type.
- 2) The *case value* must be an integer or character constant.
- 3) The *case value* can be used only inside the switch statement.
- 4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

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

Functioning of switch case statement:

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases

can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

Loops and Iterations

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language.

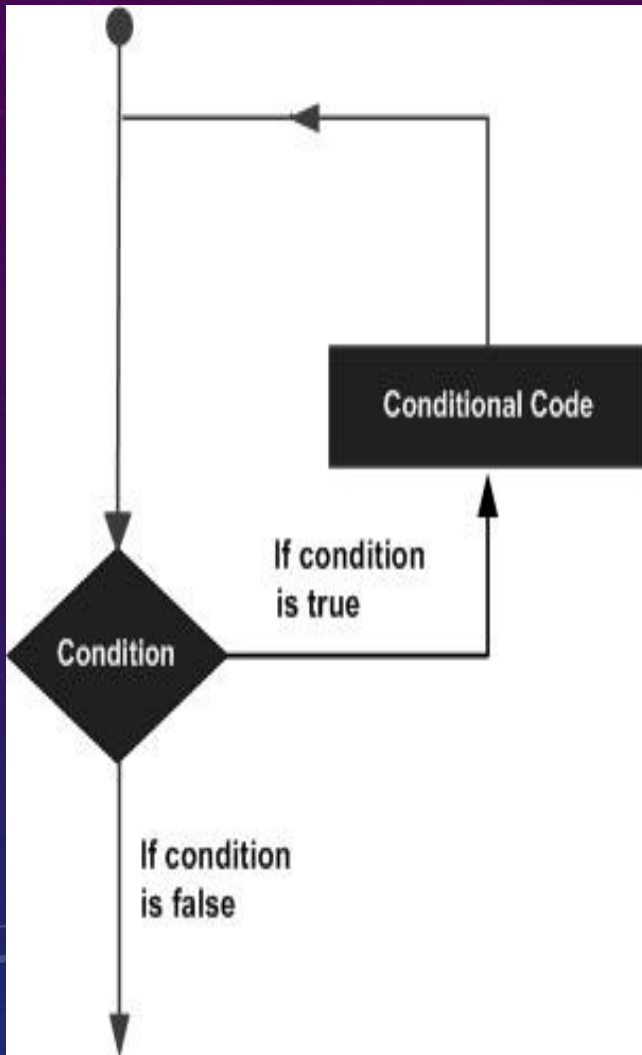
Advantage of loops in C

- It provides code reusability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of data structures (array or linked lists.)

There are types of loops in C language that is given below:

- **do while** - The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).
- **While** - The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
- **For** - The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.
- **nested** - You can use one or more loops inside any other while, for, or do..while loop.

Loops and its Functioning



Loop Control Statements--Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

- Break statement - Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
- Continue statement - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- Goto statement - Transfers control to the labeled statement.

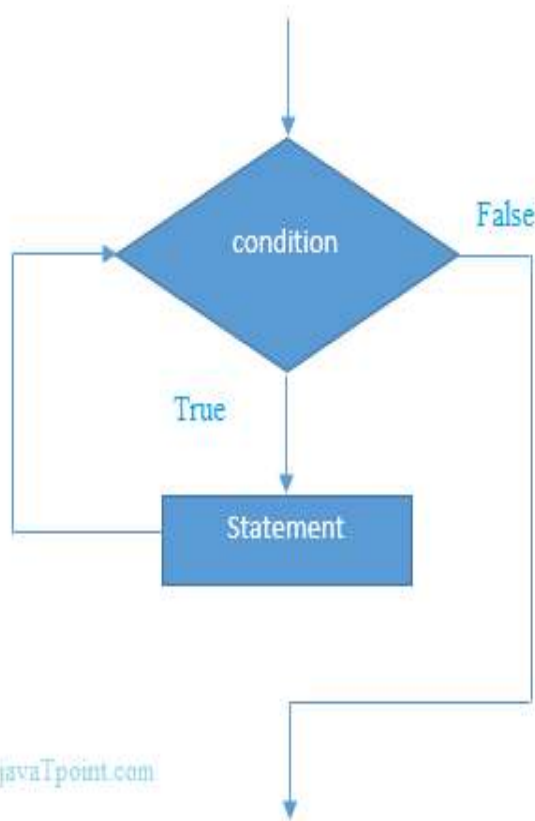
The Infinite Loop:-

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

```
#include <stdio.h>
int main ()
{
    for( ; ; )
    {
        printf("This loop will run forever.\n");
    } return 0;
}
```


Loop – While Loop

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.



```
#include<stdio.h>
int main()
{
    int i=1;
    while(i<=10){
        printf("%d \n",i);

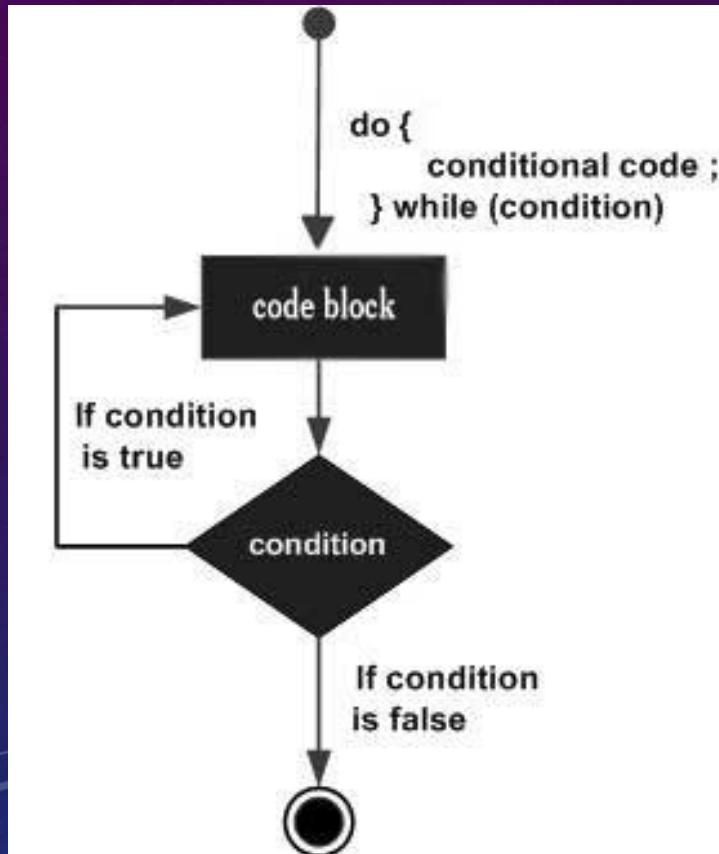
        i++;
    }
    return 0;
}
```

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

```
#include <stdio.h>
int main ()
{ /* local variable definition */
    int a = 10;
    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a); a++;
    }
    return 0;
}
```

Loop = Do-While

The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.



```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 10;
    /* do loop execution */
    do
    {
        printf("value of a: %d\n", a);
        a = a + 1;
    }
    while( a < 20 );
    return 0;
}
```

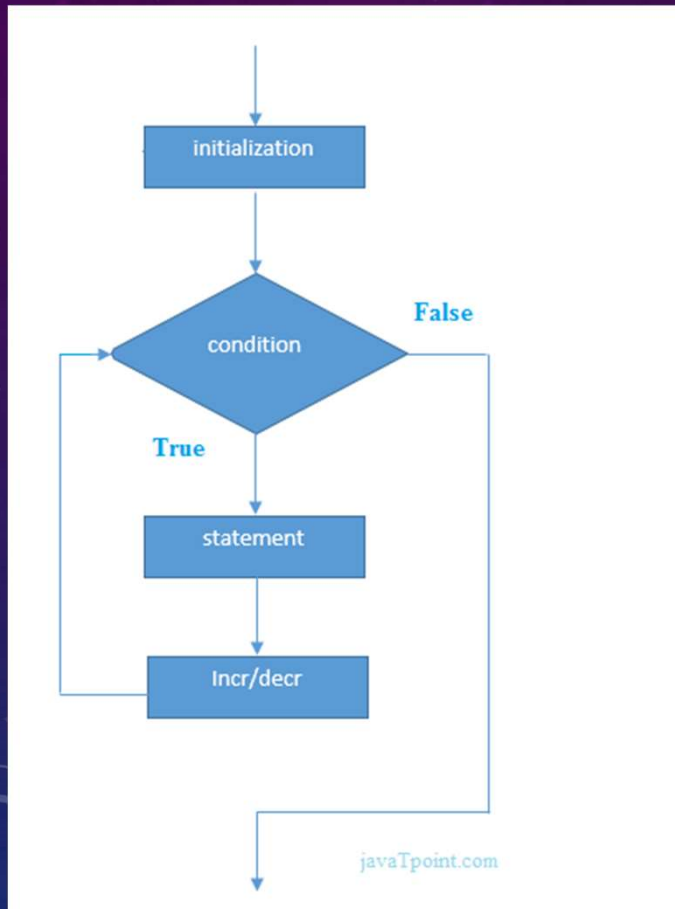
```
#include<stdio.h>
int main()
{
    int i=1,number=0;
    printf("Enter a number:");
    scanf("%d",&number);
    Do
    {
        printf("%d \n",(number*i));
        i++;
    }
    while(i<=10);
    return 0;
}
```

Loops - Difference

BASIS FOR COMPARISON	FOR	WHILE
Declaration	<pre>for(initialization; condition; iteration){ //body of 'for' loop }</pre>	<pre>while (condition) { statements; //body of loop }</pre>
Format	Initialization, condition checking, iteration statement are written at the top of the loop.	Only initialization and condition checking is done at the top of the loop.
Use	The 'for' loop used only when we already knew the number of iterations.	The 'while' loop used only when the number of iteration are not exactly known.
Condition	If the condition is not put up in 'for' loop, then loop iterates infinite times.	If the condition is not put up in 'while' loop, it provides compilation error.
Initialization	In 'for' loop the initialization once done is never repeated.	In while loop if initialization is done during condition checking, then initialization is done each time the loop iterate.
Iteration statement	In 'for' loop iteration statement is written at top, hence, executes only after all statements in loop are executed.	In 'while' loop, the iteration statement can be written anywhere in the loop.

Loop – For Loop

The **for loop in C language** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.



```
#include<stdio.h>
int main()
{
    int i=0;
    for(i=1;i<=10;i++)
    {
        printf("%d \n",i);
    }
    return 0;
}
```

```
int main()
{
    int i,j,k;
    for(i=0,j=0,k=0;i<4,k<8,j<10;i++)
    {
        printf("%d %d %d\n",i,j,k);
        j+=2;
        k+=3;
    } }
```

```
#include<stdio.h>
int main(){
    int i=1,number=0;
    printf("Enter a number: ");
    scanf("%d",&number);
    for(i=1;i<=10;i++){
        printf("%d \n",(number*i));
    }
    return 0;
}
```

```
#include <stdio.h>
int main()
{
    int a,b,c;
    for(a=0,b=12,c=23;a<2;a++)
    {
        printf("%d ",a+b+c);
    }
}
```

- `for(i<5;i++)` #include <stdio.h>
- `for(i=0;;i++)`
- `for(i = 0;i<5;i++,j=j+2)`

Loop-Nested

In programming there exists situations where you need to iterate single or a set of repetitive statement for a number of times. For example consider an example of a train. In train there are n compartments and each compartment has m seats. Ticket checker will check ticket for each passenger for each compartment.

```
#include <stdio.h>
int main ()
{ /* local variable definition */
  int i, j;
  for(i = 2; i<100; i++)
  { for(j = 2; j <= (i/j); j++)
    if(!(i%j))
      break;
    if(j > (i/j))
      printf("%d is prime\n", i);
  }
  return 0;
}
```

Break – Jump Statement

C break statement

The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. The break statement in C can be used in the following two scenarios:

- With switch case
- With loop

```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
    int i;
    for(i = 0; i<10; i++)
    {
        printf("%d ",i);
        if(i == 5)
            break;
    }
    printf("came outside of l
oop i = %d",i);
}
```

```
#include<stdio.h>
int main(){
    int i=1,j=1;//initializing a local
variable
    for(i=1;i<=3;i++){
        for(j=1;j<=3;j++){
            printf("%d &d\n",i,j);
            if(i==2 && j==2){
                break;//will break loop of j onl
y
            }
        }
    }//end of for loop
    return 0;
}
```

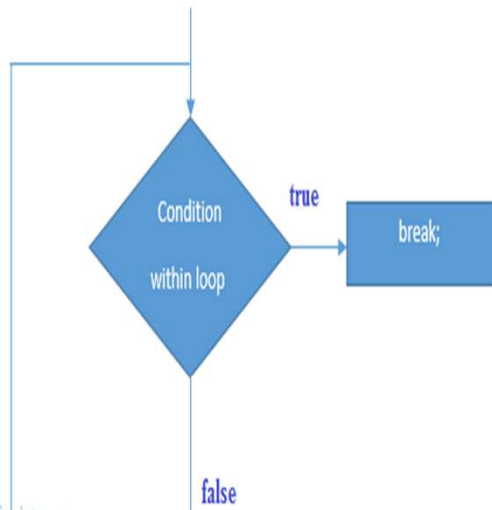
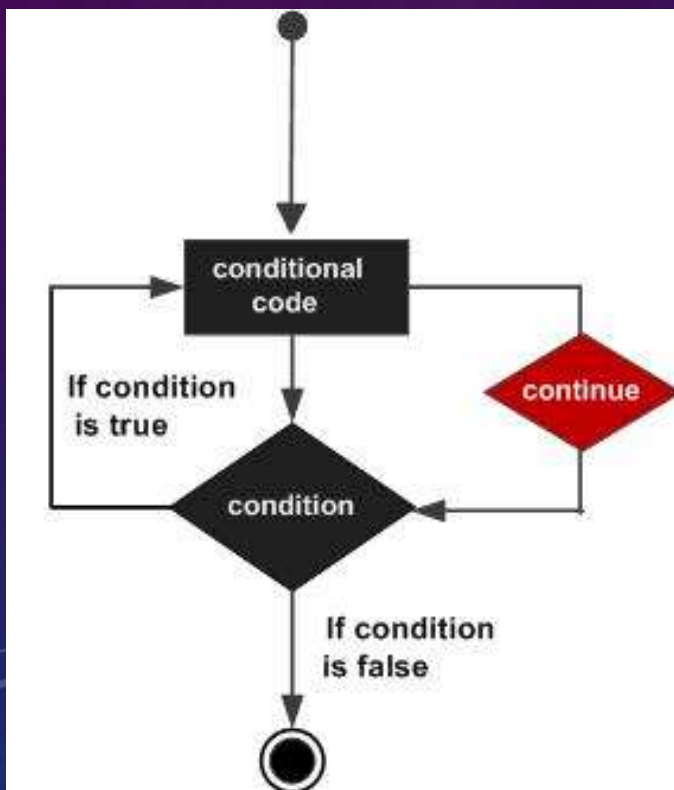


Figure: Flowchart of break statement

Continue –Jump Statement

The **continue** statement in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.



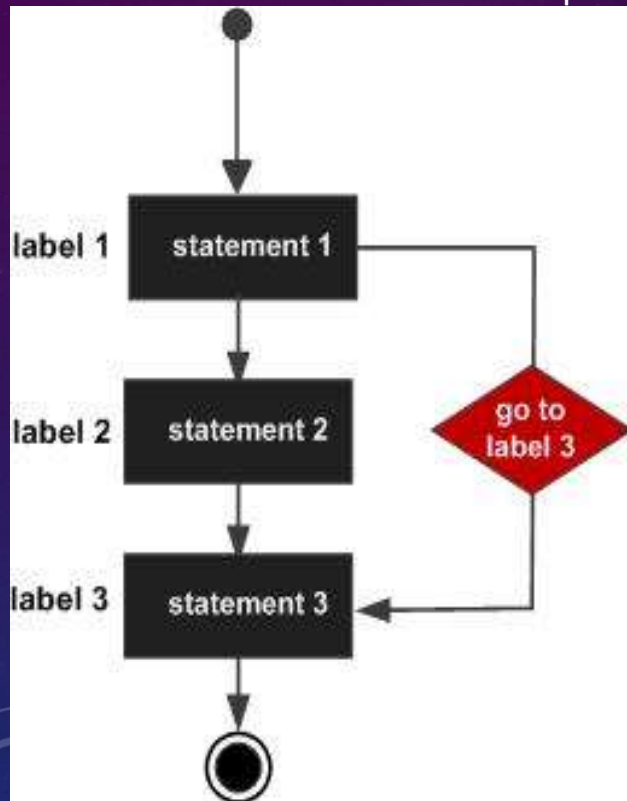
```
#include<stdio.h>
void main ()
{
    int i = 0;
    while(i!=10)
    {
        printf("%d", i);
        continue;
        i++;
    }
}
```

```
#include<stdio.h>
int main(){
    int i=1,j=1;//initializing a local variable
    for(i=1;i<=3;i++){
        for(j=1;j<=3;j++){
            if(i==2 && j==2){
                continue;//will continue loop of j only
            }
            printf("%d %d\n",i,j);
        }
    }
    //end of for loop
    return 0;
}
```

```
#include <stdio.h>
int main ()
{
    int a = 10;
    do
    {
        if( a == 15)
        {
            a = a + 1;
            continue;
        }
        printf("value of a: %d\n", a);
        a++;
    }
    while( a < 20 );
    return 0; }
```


Goto--JumpStatements

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated. The only condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. Consider the following example.



```
#include <stdio.h>
int main()
{
    int num,i=1;
    printf("Enter the number whose
table you want to print?");
    scanf("%d",&num);
table:
    printf("%d x %d = %d\n",num,i,n
um*i);
    i++;
    if(i<=10)
        goto table;
}
```

```
#include <stdio.h>
int main()
{ int i, j, k;
  for(i=0;i<10;i++)
  { for(j=0;j<5;j++)
    { for(k=0;k<3;k++)
      { printf("%d %d %d\n",i,j,k);
        if(j == 3)
        {
            goto out;
        }
      }
    }
  }
out:
  printf("came out of the loop");
}
```

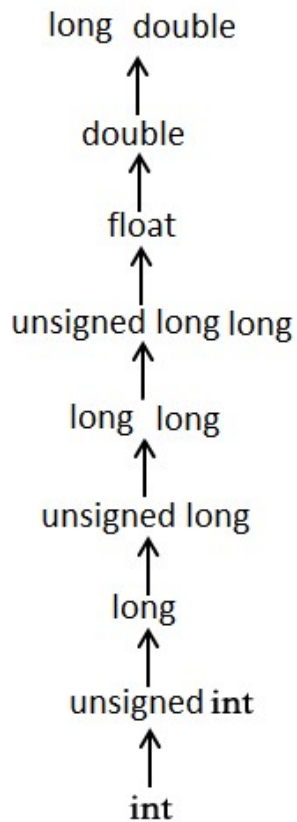
Goto – Statement

Reasons to avoid goto statement

- Though, using goto statement give power to jump to any part of program, using goto statement makes the logic of the program complex and tangled.
- In modern programming, goto statement is considered a harmful construct and a bad programming practice.
- The goto statement can be replaced in most of C program with the use of break and continue statements.
- In fact, any program in C programming can be perfectly written without the use of goto statement.
- All programmer should try to avoid goto statement as possible as they can.

TypeCasting

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**. Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).



```
#include<stdio.h>
int main()
{
    double x = 1.2;
    int sum = (int)x + 1;
    printf("sum = %d", sum);
    return 0;
}
```

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 –

```
#include<stdio.h>
int main(){
    float f= (float)9/4;
    printf("f : %f\n", f );
    return 0;
}
```

```
#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 ); }
```


Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

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.

Advantage of functions in C:

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function – Definition

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

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.

Defining a Function:

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

Function 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.

Function - Declaration

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

Function Arguments :

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

Call by value--This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by reference--This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Function - Example

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;
    /* calling a function to get max value */
    ret = max(a, b);
    printf( "Max value is : %d\n", ret );
    return 0;
}
/* function returning the max between two numbers */
int max(int num1, int num2)
{ /* local variable declaration */
    int result;
    if (num1 > num2)
        result = num1;
    else result = num2;
    return result;
}
```

Types of Functions

There are two types of functions in C programming:

- **Library Functions**: are the functions which are declared in the C header files such as `scanf()`, `printf()`, `gets()`, `puts()`, `ceil()`, `floor()` etc.
- **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value.
- function without arguments and with return value.
- function with arguments and without return value.
- function with arguments and with return value.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Function Examples:

Example for Function without argument and without return value

```
#include<stdio.h>
void printName();
void main ()
{ printf("Hello ");
  printName();
}
void printName()
{
  printf("Nishant");
}

#include<stdio.h>
void sum();
void main()
{
  printf("\nNishant Want to
  calculate the sum:");
  sum();
}
void sum()
{
  int a,b;
  printf("\nEnter two numbers");
  scanf("%d %d",&a,&b);
  printf("The sum is %d",a+b);
}
```

Example for Function without argument and with return value

```
#include<stdio.h>
int sum();
void main()
{
  int result;
  printf("\nNishantwant to calculate the sum of two numbers:");

  result = sum();
  printf("%d",result);
}
int sum()
{
  int a,b;
  printf("\nEnter two numbers");
  scanf("%d %d",&a,&b);
  return a+b;
}
```

```
#include<stdio.h>
int sum();
void main()
{ printf(" to calculate the area
of the square\n");
  float area = square();
  printf("The area of the squa
re: %f\n",area);
} int square()
{ float side;
  printf("Enter the length of t
he side in meters: ");
  scanf("%f",&side);
  return side * side;
}
```


Function : Examples

Function with argument and without return value.

```
#include<stdio.h>
void sum(int, int);
void main()
{   int a,b,result;
    printf("\nGoing to cal the sum of two num:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

```
#include<stdio.h>
void average(int, int, int, int, int);
void main()
{
    int a,b,c,d,e;
    printf("\nGoing to calculate the average of five numbers:");
    printf("\nEnter five numbers:");
    scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);

    average(a,b,c,d,e);
}
void average(int a, int b, int c, int d, int e)
{
    float avg;
    avg = (a+b+c+d+e)/5;
    printf("The average of given five numbers : %f",avg);
}
```

Function - Examples

Function with argument and with return value.

```
#include<stdio.h>
int sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two number
s:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    result = sum(a,b);
    printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
    return a+b;
}
```

```
#include<stdio.h>
int even_odd(int);
void main()
{
    int n,flag=0;
    printf("\n to check a num is even or odd");
    printf("\nEnter the number: ");
    scanf("%d",&n);
    flag = even_odd(n);
    if(flag == 0)
    { printf("\nThe number is odd");
    }
    else
    { printf("\nThe number is even");
    }
}
int even_odd(int n)
{ if(n%2 == 0)
    {
        return 1;
    }
    else
    { return 0;
    } }
}
```

Function - Call by value and Reference

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.

Call by value in C:-

- 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.

Call by reference in C:-

- 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.

Call by Value - Example

```
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);

    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
    swap(a,b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
}
```

Call by Reference - Example

```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside f
unction num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside fun
ction num=%d \n", *num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n
", x);
    change(&x);//passing reference in f
unction
    printf("After function call x=%d \n",
x);
    return 0;
}
```

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printi
ng the value of a and b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values
of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Form
al parameters, a = 20, b = 10
}
```

Call by Value and Reference

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Function - Recursion

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand. Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1
```

javaTpoint.com

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

Examples of Recursion :--

- Factorial.
- Fibonacci series.

Recursion -- Examples

```
#include <stdio.h>
int fact (int);
int main()
{ int n,f;
  printf("Enter the number to calculate factorial");
  scanf("%d",&n);
  f = fact(n);
  printf("factorial = %d",f);
}
int fact(int n)
{
  if (n==0)
  {
    return 0;
  }
  else if ( n == 1)
  {
    return 1;
  }
  else
  {
    return n*fact(n-1);
  } }
}
```

Fibonacci series

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
  int n,f;
  printf("Enter the value of n?");
  scanf("%d",&n);
  f = fibonacci(n);
  printf("%d",f);
}
int fibonacci (int n)
{
  if (n==0)
  {
    return 0;
  }
  else if (n == 1)
  {
    return 1;
  }
  else
  {
    return fibonacci(n-1)+fibonacci(n-2);
  } }
}
```

C- Array

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

Properties of Array:--

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array:--

- 1) **Code Optimization:** Less code to access the data.
- 2) **Ease of traversing:** By using the for loop, we can retrieve the elements of an array easily.
- 3) **Ease of sorting:** To sort the elements of the array, we need a few lines of code only.
- 4) **Random Access:** We can access any element randomly using the array.

Disadvantage of C Array:--

1) **Fixed Size:**

Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Array - Introduction



Step by step working of an Array:

Declaration of C Array

- data_type array_name[array_size];
e.g.---`int marks[5];`

Initialization of C Array

- `marks[0]=80;`//initiztn of array
- `marks[1]=60;`
- `marks[2]=70;`
- `marks[3]=85;`
- `marks[4]=75;`

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

```
int main(){
```

```
int i=0;
```

```
int marks[5];//declartn of array
```

```
marks[0]=80;//initiliztn of array
```

```
marks[1]=60;
```

```
marks[2]=70;
```

```
marks[3]=85;
```

```
marks[4]=75;
```

```
//traversal of array
```

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

```
printf("%d \n",marks[i]);
```

```
//end of for loop
```

```
return 0;
```

```
}
```

Declaration with Initialization.

```
int marks[5]={20,30,40,50,60}; no req. to define size
```

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

```
int main(){
```

```
int i=0;
```

```
int marks[5]={20,30,40,50,60};//declaration and initialization of array
```

```
//traversal of array
```

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

```
printf("%d \n",marks[i]);
```

```
}
```

```
return 0;
```

```
}
```

Accessing Array Elements

```
printf("%d \n",marks[0]);
```

```
printf("%d \n",marks[1]);
```

```
printf("%d \n",marks[3]);
```

C- Array Program

#include<stdio.h> Sorting of an array

```
void main ()
{
    int i, j, temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};

    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n", a[i]);
    }
}
```

Largest and Second Largest

#include<stdio.h>

```
void main ()
{
    int arr[100], i, n, largest, sec_largest;
    printf("Enter the size of the array?");
    scanf("%d", &n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0; i<n; i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
            largest = arr[i];
        }
        else if (arr[i]>sec_largest && arr[i]!=largest)
        {
            sec_largest=arr[i];
        }
    }
    printf("largest = %d, second largest = %d", largest, sec_largest);
}
```

Array –2D

2-Dimensional Array

- The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the **collection of rows and columns**. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.
- 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].

Declaration of 2-Dimensional Array

data_type array_name[rows][columns];

- **int** twodimen[4][3];

Initialization of 2D Array

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

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array.

Accessing 2-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.

- `printf("a[%d][%d] = %d\n", i,j, a[i][j]);`

2D – Array Programs

```
#include<stdio.h>
int main(){
int i=0,j=0;
int arr[4][3]={1,2,3},{2,3,4},{3,4,5},{4,5,6}};

//traversing 2D array
for(i=0;i<4;i++){
for(j=0;j<3;j++){
printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
}
}
return 0;
}
```

Nd array

```
#include <stdio.h>
int 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] );
}
}
return 0;
}
```

Storing elements

```
#include <stdio.h>
void main ()
{
int arr[3][3],i,j;
for (i=0;i<3;i++)
{
for (j=0;j<3;j++)
{
printf("Enter a[%d][%d]: ",i,j);

scanf("%d",&arr[i][j]);
}
printf("\n printing the elements .
...\n");
for(i=0;i<3;i++)
{
printf("\n");
for (j=0;j<3;j++)
{
printf("%d\t",arr[i][j]);
} } }
}
```

C - String

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as `char s[10]`, the character `s[10]` is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

- By char array.
e.g.----**char** ch[10]={'n', 'i', 's', 'h', 'a', 'n', 't', '\0'};
- By string literal.
e.g.--- **char** ch[]="nishant";

Difference between char array and string literal

There are two main differences between char array and literal.

- We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
- The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

C- String

Traversing String:

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string

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

```
void main ()
```

```
{
```

```
    char s[11] = "nishant";
```

```
    int i = 0;
```

```
    int count = 0;
```

```
    while(i<11)
```

```
    {
```

```
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
```

```
        { count ++;
```

```
        }
```

```
        i++;
```

```
    }
```

```
    printf("The number of vowels %d",count);
```

```
}
```

- By using the null character.

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

```
void main ()
```

```
{
```

```
    char s[11] = "nishant";
```

```
    int i = 0;
```

```
    int count = 0;
```

```
    while(s[i] != NULL)
```

```
    {
```

```
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' |  
| s[i] == 'o')
```

```
        {
```

```
            count ++;
```

```
        }
```

```
        i++;
```

```
    }
```

```
    printf("The number of vowels %d",count);
```

```
}
```


C- String

Accepting string as the input

Not with space

```
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%s",s);
    printf("You entered %s",s);
}
```

With Space

```
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");

    scanf("%[^\n]s",s);
    printf("You entered %s",s);
}
```

However, there are the following points which must be noticed while entering the strings by using scanf. The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data. Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

String – Functions(gets)

C-- gets() functions :--

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

- C gets() function--The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

```
#include<stdio.h>
void main ()
{
    char s[30];
    printf("Enter the string? ");
    gets(s);
    printf("You entered %s",s);
}
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read.

```
#include<stdio.h>
void main()
{
    char str[20];
    printf("Enter the string? ");
    fgets(str, 20, stdin);
    printf("%s", str);
}
```

String function--put

C puts() function

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

```
#include<stdio.h>
#include <string.h>
int main(){
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name); //displays string
return 0;
}
```


C String Functions

No.	Function	Description
1)	<code>strlen(string_name)</code>	returns the length of string name.
2)	<code>strcpy(destination, source)</code>	copies the contents of source string to destination string.
3)	<code>strcat(first_string, second_string)</code>	concatenates or joins first string with second string. The result of the string is stored in first string.
4)	<code>strcmp(first_string, second_string)</code>	compares the first string with second string. If both strings are same, it returns 0.
5)	<code>strrev(string)</code>	returns reverse string.
6)	<code>strlwr(string)</code>	returns string characters in lowercase.
7)	<code>strupr(string)</code>	returns string characters in uppercase.

C- String Functions Examples

1- String Length: strlen() function:

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
#include<stdio.h>
#include <string.h>
int main(){
char ch[20]={'n', 'i', 's', 'h', 'a', 'n', 't', '\0'};
printf("Length of string is: %d",strlen(ch));
return 0;
}
```

2- Copy String: strcpy() function:

```
#include<stdio.h>
#include <string.h>
int main(){
char ch[20]={'n', 'i', 's', 'h', 'a', 'n', 't', '\0'};
strcpy(ch2,ch);
printf("Value of second string is: %s",ch2);
return 0;
}
```

C- String Functions Examples

String Concatenation: strcat()

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

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

```
int main()
{
    char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
    char ch2[10]={'c', '\0'};
    strcat(ch,ch2);
    printf("Value of first string is: %s",ch);
    return 0;
}
```

Compare String: strcmp():

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str1[20],str2[20];
    printf("Enter 1st string: ");
    gets(str1);//reads string from console
    printf("Enter 2nd string: ");
    gets(str2);
    if(strcmp(str1,str2)==0)
        printf("Strings are equal");
    else
        printf("Strings are not equal");
    return 0;
}
```


C- String Functions Examples

Reverse String: strrev():

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[20];
    printf("Enter string: ");
    gets(str);//reads string from console
    printf("String is: %s",str);
    printf("\nReverse String is: %s",strrev(str));
    return 0;
}
```

String Lowercase: strlwr():

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[20];
    printf("Enter string: ");
    gets(str);//reads string from console
    printf("String is: %s",str);
    printf("\nLower String is: %s",strlwr(str));
    return 0;
}
```

C- String Functions Examples

String Uppercase: strupr()

The `strupr(string)` function returns string characters in uppercase. Let's see a simple example of `strupr()` function.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[20];
    printf("Enter string: ");
    gets(str);//reads string from console
    printf("String is: %s",str);
    printf("\nUpper String is: %s",strupr(str));

    return 0;
}
```

String strstr():

The `strstr()` function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

- **string:** It represents the full string from where substring will be searched.
- **match:** It represents the substring to be searched in the full string.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[100]="Hello how are you";
    char *sub;
    sub=strstr(str,"how");
    printf("\nSubstring is: %s",sub);
    return 0;
}
```

C- Math Functions

C Programming allows us to perform mathematical operations through the functions defined in `<math.h>` header file. The `<math.h>` header file contains various methods for performing mathematical operations such as `sqrt()`, `pow()`, `ceil()`, `floor()` etc.

No.	Function	Description
1)	<code>ceil(number)</code>	rounds up the given number. It returns the integer value which is greater than or equal to given number.
2)	<code>floor(number)</code>	rounds down the given number. It returns the integer value which is less than or equal to given number.
3)	<code>sqrt(number)</code>	returns the square root of given number.
4)	<code>pow(base, exponent)</code>	returns the power of given number.
5)	<code>abs(number)</code>	returns the absolute value of given number.

C- Math Example

```
#include<stdio.h>
#include <math.h>
int main(){
printf("\n%f",ceil(3.6));
printf("\n%f",ceil(3.3));
printf("\n%f",floor(3.6));
printf("\n%f",floor(3.2));
printf("\n%f",sqrt(16));
printf("\n%f",sqrt(7));
printf("\n%f",pow(2,4));
printf("\n%f",pow(3,3));
printf("\n%d",abs(-12));
return 0;
}
```

C-- Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

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. Let's start learning them in simple and easy steps. 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.

Example---

```
#include <stdio.h>
int main ()
{
int var1;
char var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
return 0;
}
```

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.

Pointer - Working

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;
}
```

Advantage of pointer

- 1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- 2) We can **return multiple values from a function** using the pointer.
- 3) It makes you able to **access any memory location** in the computer's memory.

C - Pointer

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

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.

```
#include <stdio.h>
int main ()
{
int *ptr = NULL;
printf("The value of ptr is : %x\n", ptr );
return 0;
}
```

Swap a number using pointer

```
#include<stdio.h>
int main(){
int a=10,b=20,*p1=&a,*p2=&b;

printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);

return 0;
}
```

Pointer - Arithmetic

Incrementing a Pointer:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{ int var[] = {10, 100, 200};

int i, *ptr; /* let us have array address in
pointer */

ptr = var;
for ( i = 0; i < MAX; i++)
{
printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, *ptr ); /*
mv to the next locatn */

ptr++;
}
return 0;
}
```

Pointer Comparisons:






```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr; /* let us have address of the first element in pointer */
ptr = var; i = 0;
while ( ptr <= &var[MAX - 1] )
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr ); /* point to the previous
location */
ptr++;
i++;
}
return 0;
}
```

C-illegal Expressions with pointers

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

Sr.No.	Concept & Description
1	Pointer arithmetic  There are four arithmetic operators that can be used in pointers: ++, --, +, -
2	Array of pointers  You can define arrays to hold a number of pointers.
3	Pointer to pointer  C allows you to have pointer on a pointer and so on.
4	Passing pointers to functions in C  Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5	Return pointer from functions in C  C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

C – Pointer to Pointer

A pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.

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

```
#include<stdio.h>
void main ()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a; // pointer p is pointing to the address of a
    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
    printf("address of a: %x\n",p); // Address of a will be printed
    printf("address of p: %x\n",pp); // Address of p will be printed
    printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10 will be printed
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer stored at pp
}
```

C – Array Pointer

In Array pointer, we can store address of a variable in an array. There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available.

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
    int var[] = {10, 100, 200};
    int i;
    for (i = 0; i < MAX; i++)
    {
        printf("Value of var[%d] = %d\n", i, var[i] );
    }
    return 0;
}
```

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
    int var[] = {10, 100, 200};
    int i, *ptr[MAX];
    for ( i = 0; i < MAX; i++)
    {
        ptr[i] = &var[i]; /* assign the
                           address of integer. */
    }
    for ( i = 0; i < MAX; i++)
    {
        printf("Value of var[%d] = %d\n", i, *ptr[i] );
    }
    return 0;
}
```

```
#include <stdio.h>
const int MAX = 4;
int main ()
{
    char *names[] = { "Nishant", "Sam",
                     "Nik", "Devil" };
    int i = 0;
    for ( i = 0; i < MAX; i++)
    {
        printf("Value of names[%d] = %s\n",
              i, names[i] );
    }
    return 0;
}
```

C- Passing Pointer to a Function And Returning also

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

```
#include <stdio.h>
#include <time.h>
void getSeconds(unsigned long *par);
int main ()
{
    unsigned long sec;
    getSeconds( &sec ); /* print the actual value */
    printf("Number of seconds: %ld\n", sec );
    return 0;
}
void getSeconds(unsigned long *par)
{
    /* get the current number of seconds */
    *par = time( NULL );
    return;
}
```

```
#include <stdio.h> /* function declaration */
double
getAverage(int *arr, int size);
int main ()
{
    /* an int array with 5 elements */
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg; /* pass pointer to the array as an argument */
    avg = getAverage( balance, 5 ); /* output the returned
    value */
    printf("Average value is: %f\n", avg );
    return 0;
}
double getAverage(int *arr, int size)
{
    int i, sum = 0;
    double avg;
    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    avg = (double)sum / size;
    return avg;
}
```


C- Passing Pointer to a Function And Returning also

C also allows to return a pointer from a function.

```
#include <stdio.h>
#include <time.h> /* function to generate and return random numbers. */
int * getRandom( )
{
    static int r[10]; int i; /* set the seed */
    srand( (unsigned)time( NULL ) );
    for ( i = 0; i < 10; ++i)
    {
        r[i] = rand();
        printf("%d\n", r[i] );
    }
    return r;
}
/* main function to call above defined function */
int main ()
{
    /* a pointer to an int */
    int *p; int i;
    p = getRandom();
```

```
    for ( i = 0; i < 10; i++ )
    {
        printf("(p + [%d]) : %d\n", i, *(p + i) );
    }
    return 0;
}
```

C - Structure

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

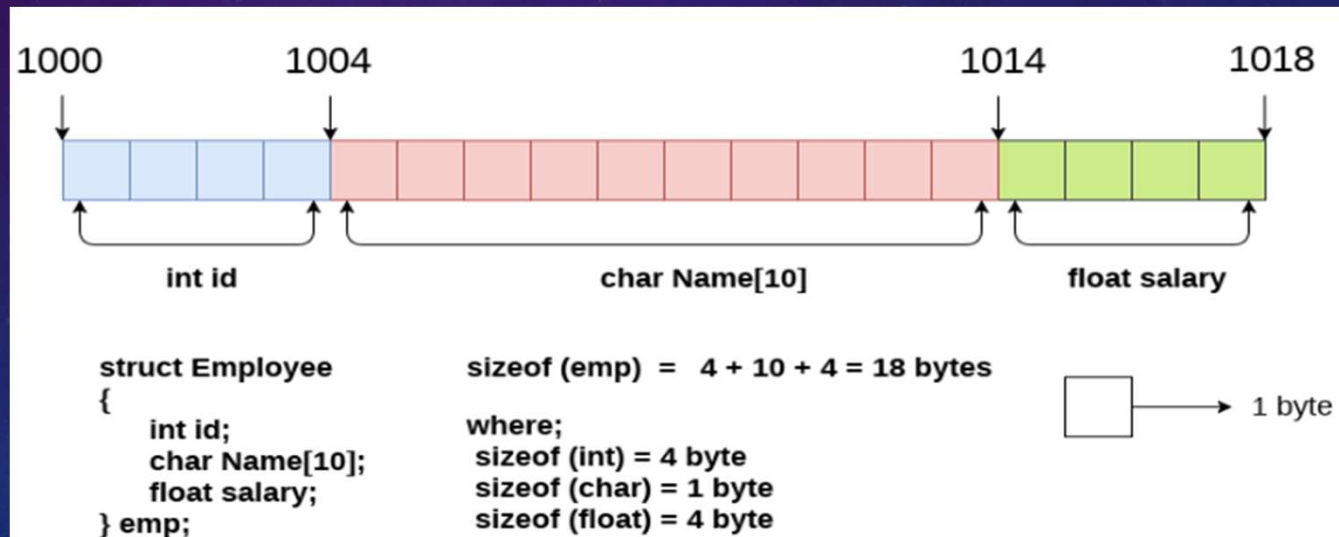
For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.

- In structure, data is stored in form of **records**.

Defining a Structure:

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member.

```
struct employee
{ int id;
  char name[20];
  float salary;
};
```



Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure.

C - Structure Declaration

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

- By struct keyword within main() function

```
struct employee
{ int id;
  char name[50];
  float salary;
};
```

```
struct employee e1, e2;
```

- By declaring a variable at the time of defining the structure.

```
struct employee
{ int id;
  char name[50];
  float salary;
}e1,e2;
```

Which approach is good

- If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.
- If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

C- Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type.

There are two ways to access structure members:

- By . (member or dot operator)
- By -> (structure pointer operator)

```
#include<stdio.h>
#include <string.h>
struct employee
{
    int id;
    char name[50];
}
e1; //declaring e1 variable for structure
int main( )
{
    //store first employee information
    e1.id=101;
    strcpy(e1.name, "Nishant");//copying string into char array
    //printing first employee information
    printf( "employee 1 id : %d\n", e1.id);
    printf( "employee 1 name : %s\n", e1.name);
    return 0;
}
```

```
#include<stdio.h>
#include <string.h>
struct employee
{
    int id; char name[50]; float salary;
}e1,e2; //declaring e1 and e2 variables for structure
int main( )
{
    //store first employee information
    e1.id=101;
    strcpy(e1.name, "Nishant");//copying string into char array
    e1.salary=56000;
    //store second employee information
    e2.id=102;
    strcpy(e2.name, "Devil");
    e2.salary=126000;
    //printing first employee information
    printf( "employee 1 id : %d\n", e1.id);
    printf( "employee 1 name : %s\n", e1.name);
    printf( "employee 1 salary : %f\n", e1.salary);
    //printing second employee information
    printf( "employee 2 id : %d\n", e2.id);
    printf( "employee 2 name : %s\n", e2.name);
    printf( "employee 2 salary : %f\n", e2.salary);
    return 0; }
```

C- Structure Examples

```
#include <stdio.h>
#include <string.h>
struct Books
{ char title[50];
  char author[50];
  char subject[100];
  int book_id;
};
int main( )
{
    struct Books Book1; /* Declare Book1 of type Book */
    struct Books Book2; /* Declare Book2 of type Book */
    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nishant");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;
    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Abc");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;
```

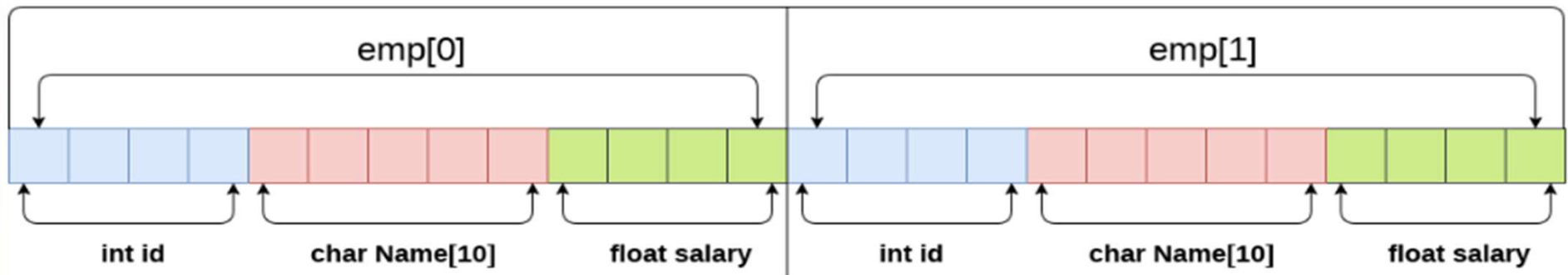
```
/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

/* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0;
}
```

C - Array Structure

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

`sizeof (emp) = 4 + 5 + 4 = 13 bytes`

`sizeof (emp[2]) = 26 bytes`

C – Array Structure

```
#include<stdio.h>
struct student
{
    char name[20];
    int id;
    float marks;
};

void main()
{
    struct student s1,s2,s3;
    int dummy;
    printf("Enter the name, id, and marks of student 1 ");
    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
    scanf("%c",&dummy);
    printf("Enter the name, id, and marks of student 2 ");
    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
    scanf("%c",&dummy);
    printf("Enter the name, id, and marks of student 3 ");
    scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
    scanf("%c",&dummy);
    printf("Printing the details...\n");
    printf("%s %d %f\n",s1.name,s1.id,s1.marks);
    printf("%s %d %f\n",s2.name,s2.id,s2.marks);
    printf("%s %d %f\n",s3.name,s3.id,s3.marks);
}
```

(Manually insertion)→

Using Array

```
#include<stdio.h>
#include <string.h>
struct student{
    int rollno;
    char name[10];
};

int main(){
    int i;
    struct student st[5];
    printf("Enter Records of 5 students");
    for(i=0;i<5;i++)
    {
        printf("\nEnter Rollno:");
        scanf("%d",&st[i].rollno);
        printf("\nEnter Name:");
        scanf("%s",&st[i].name);
    }
    printf("\nStudent Information List:");
    for(i=0;i<5;i++){
        printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
    }
    return 0;
}
```

C—Struture With Function

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

```
#include <stdio.h>
#include <string.h>
struct Books
{ char title[50]; char author[50]; char subject[100];
  int book_id; };
/* function declaration */
void printBook( struct Books book );
int main( )
{ struct Books Book1; /* Declare Book1 of type Book */
  struct Books Book2; /* Declare Book2 of type Book */
  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nishant");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;
  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "DEvil");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;
  /* print Book1 info */
  printBook( Book1 );
```

```
/* Print Book2 info */
printBook( Book2 );
return 0;
}
void printBook( struct Books book )
{
  printf( "Book title : %s\n", book.title);
  printf( "Book author : %s\n", book.author);
  printf( "Book subject : %s\n", book.subject);
  printf( "Book book_id : %d\n", book.book_id);
}
```

C--Pointers to Structures

To access the members of a structure using a pointer to that structure, you must use the → operator.

```
#include <stdio.h>
#include <string.h>
struct Books
{
char title[50]; char author[50]; char subject[100];
int book_id;
};
/* function declaration */
void printBook( struct Books *book );
int main( )
{
struct Books Book1; /* Declare Book1 of type Book */
struct Books Book2; /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nishant");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Devil");
```

```
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;
/* print Book1 info by passing address of Book1 */
```

```
printBook( &Book1 );
/* print Book2 info by passing address of Book2 */
```

```
printBook( &Book2 );
return 0;
}
void printBook( struct Books *book )
{
printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id); }
```


C – Nested Structure

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee.

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

```
struct address
```

```
{
```

```
    char city[20];
```

```
    int pin;
```

```
    char phone[14];
```

```
};
```

```
struct employee
```

```
{
```

```
    char name[20];
```

```
    struct address add;
```

```
};
```

```
void main ()
```

```
{
```

```
    struct employee emp;
```

```
    printf("Enter employee information?\n");
```

```
    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
```

```
    printf("Printing the employee information....\n");
```

```
    printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone); }
```

The structure can be nested in the following ways.

- By separate structure
- By Embedded structure

C- Types of Nested Structure

Separate Structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member.

```
struct Date
{
    int dd;
    int mm;
    int yyyy;
};
struct Employee
{
    int id;
    char name[20];
    struct Date doj;
}emp1;
```

Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures.

```
struct Employee
{
    int id;
    char name[20];
    struct Date
    {
        int dd;
        int mm;
        int yyyy;
    }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.

Nested Structure Example

```
#include <stdio.h>
#include <string.h>
struct Employee
{
    int id; char name[20];
    struct Date
    {
        int dd; int mm; int yyyy;
    }doj;
}e1;
int main( )
{ //storing employee information
  e1.id=101;
  strcpy(e1.name, "Nishant");//copying string into char array
  e1.doj.dd=10;
  e1.doj.mm=11;
  e1.doj.yyyy=2014;
  //printing first employee information
  printf( "employee id : %d\n", e1.id);
  printf( "employee name : %s\n", e1.name);
  printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.doj.yyyy);
  return 0;
}
```


C – Structure to a Function.

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. To pass the structure variable employee to a function display() which is used to display the details of an employee.

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

```
struct address
```

```
{  
    char city[20];  
    int pin;  
    char phone[14];  
};
```

```
struct employee
```

```
{  
    char name[20];  
    struct address add;  
};
```

```
void display(struct employee);
```

```
void main ()
```

```
{  
    struct employee emp;  
    printf("Enter employee information?\n");  
    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);  
    display(emp);  
}
```

```
void display(struct employee emp)
```

```
{  
    printf("Printing the details....\n");  
    printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);  
}
```

C- Union

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Like structure, **Union in c language** is *a user-defined data type* that is used to store the different type of elements.

At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.

Advantage of union over structure

- It **occupies less memory** because it occupies the size of the largest member only.

Disadvantage of union over structure

- Only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The **union** keyword is used to define the union.

e.g.

union employee

```
{ int id;  
  char name[50];  
  float salary;  
};
```

Union - Example

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

```
union employee
{ int id;
  char name[50];
}e1; //declaring e1 variable for union
int main( )
{
  //store first employee information
  e1.id=101;
  //copying string into char array
  strcpy(e1.name, "Nishant");
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  return 0;
}
```

Size checking for union

```
#include <stdio.h>
#include <string.h>
union Data
{
  int i;
  float f;
  char str[20];
};
int main( )
{
  union Data data;
  printf( "Memory size occupied by data :
  %d\n", sizeof(data)); return 0; }
```


C – Union(Accessing Element)

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type.

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i; float f; char str[20];
};
int main( )
{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

C - typedef

The C programming language provides a keyword called **typedef**, which you can use to give a type a new name.

e.g.---typedef unsigned char BYTE;

After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char**, for example.

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences –

typedef is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.

typedef interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.

```
#include <stdio.h>
#include <string.h>
typedef struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id; } Book;
int main( )
{
```

```
    Book book;
    strcpy( book.title, "C Programming");
    strcpy( book.author, "Nishant");
    strcpy( book.subject, "C Programming Tutorial");
    book.book_id = 6495407;
    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);
    return 0;
}
```

C - Input/Output

- **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.
- **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files:

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

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

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions:

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen.

C – Input/Output

```
#include <stdio.h>
int main( )
{
    int c;
    printf( "Enter a value :");
    c = getchar( );
    printf( "\nYou entered: ");
    putchar( c );
    return 0;
}
```

The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char *s)** function writes the string 's' and 'a' trailing newline to **stdout**.

NOTE: Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

```
#include <stdio.h>
int main( )
{
    char str[100];
    printf( "Enter a value :");
    gets( str );
    printf( "\nYou entered: ");
    puts( str );
    return 0;
}
```

When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays.

C – File Handling

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

File Handling -- Opening File

You can use the **fopen()** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream.

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "**c://some_folder/some_file.ext**".
- The mode in which the file is to be opened. It is a string.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

File Handling – Opens

```
#include<stdio.h>
void main( )
{
    FILE *fp ;
    char ch ;
    fp = fopen("file_handle.c","r") ;
    while ( 1 )
    {
        ch = fgetc ( fp ) ;
        if ( ch == EOF )
            break ;
        printf("%c",ch) ;
    }
    fclose (fp ) ;
}
```

Closing a File

To close a file, use the `fclose()` function.

The **`fclose(-)`** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **`stdio.h`**.

```
int fclose( FILE *fp );
```

C fprintf() and fscanf():

Writing File : `fprintf()` function: The `fprintf()` function is used to write set of characters into file. It sends formatted output to a stream.

```
#include <stdio.h>
main()
{
    FILE *fp;
    fp = fopen("file.txt", "w");//opening file
    fprintf(fp, "Hello file by fprintf...\n");//writing data into file
    fclose(fp);//closing file
}
```

File---fputc and fgetc

Writing File : fputc() function.

The fputc() function is used to write a single character into file. It outputs a character to a stream.

int fputc(int c, FILE *stream)

```
#include <stdio.h>
main()
{
    FILE *fp;
    fp = fopen("file1.txt", "w");//opening file
    fputc('a',fp);//writing single character into file
    fclose(fp);//closing file
}
```

Reading File : fgetc() function.

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

int fgetc(FILE *stream)

```
#include<stdio.h>
#include<conio.h>
void main(){
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("myfile.txt","r");

    while((c=fgetc(fp))!=EOF){
        printf("%c",c);
    }
    fclose(fp);
    getch();
}
```

C- fputs and fgets

The fputs() and fgets() in C programming are used to write and read string from stream.

Writing File :

fputs() function--The fputs() function writes a line of characters into file. It outputs string to a stream.

```
int fputs(const char *s, FILE *stream)
```

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

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

```
void main(){
```

```
FILE *fp;
```

```
clrscr();
```

```
fp=fopen("myfile2.txt","w");
```

```
fputs("hello c programming",fp);
```

```
fclose(fp);
```

```
getch();
```

```
}
```

Reading File :

fgets() function--The fgets() function reads a line of characters from file. It gets string from a stream.

```
char* fgets(char *s, int n, FILE *stream)
```

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

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

```
void main(){
```

```
FILE *fp;
```

```
char text[300];
```

```
clrscr();
```

```
fp=fopen("myfile2.txt","r");
```

```
printf("%s",fgets(text,200,fp));
```

```
fclose(fp);
```

```
getch();
```

```
}
```


File – Seek, Rewind

C fseek() function:

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

int fseek(**FILE** *stream, **long int** offset, **int** whence)

There are 3 constants used in the fseek() function for whence: SEEK_SET, SEEK_CUR and SEEK_END.

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

```
void main(){
```

```
    FILE *fp;
```

```
    fp = fopen("myfile.txt","w+");
```

```
    fputs("My World", fp);
```

```
    fseek( fp, 7, SEEK_SET );
```

```
    fputs("Nishant", fp);
```

```
    fclose(fp);
```

```
}
```

C rewind() function:

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax-----void rewind(**FILE** *stream)

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

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

```
void main(){
```

```
    FILE *fp;
```

```
    char c;
```

```
    clrscr();
```

```
    fp=fopen("file.txt","r");
```

```
    while((c=fgetc(fp))!=EOF)
```

```
{
```

```
    printf("%c",c);
```

```
}
```

```
    rewind(fp); //moves the file pointer at beginning of the file
```

```
    while((c=fgetc(fp))!=EOF){
```

```
    printf("%c",c);
```

```
}
```

```
    fclose(fp);
```

```
    getch(); }
```

As you can see, rewind() function moves the file pointer at beginning of the file that is why "this is simple text" is printed 2 times. If you don't call rewind() function, "this is simple text" will be printed only once.

C-- ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file. Syntax-----**long int** ftell(**FILE** *stream).

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

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

```
void main (){
```

```
    FILE *fp;
```

```
    int length;
```

```
    clrscr();
```

```
    fp = fopen("file.txt", "r");
```

```
    fseek(fp, 0, SEEK_END);
```

```
    length = ftell(fp);
```

```
    fclose(fp);
```

```
    printf("Size of file: %d bytes", length);
```

```
    getch();
```

```
}
```

C --Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

C- Macros

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

- Object-like Macros.---The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. E.g.--#define PI 3.14
- Function-like Macros----The function-like macro looks like function call. For example: #define MIN(a,b) ((a)<(b)?(a):(b))

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

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

```
#include<stdio.h>
int main()
{
    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("STDC :%d\n", __STDC__ );
    return 0;
}
```

Preprocessor-- # include and # define

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

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

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

C #define:

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

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
    printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Preprocessor- #undef

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

```
#include <stdio.h>
#define PI 3.14
#undef PI
main()
{
    printf("%f",PI);
}
```

Output-----Error

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

```
#include <stdio.h>
#define number 15
int square=number*number;
#undef number
main() {
    printf("%d",square);
}
```

Here, we are defining and undefining number variable. But before being undefined, it was used by square variable.

Preprocessor-- #ifdef and #ifndef

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

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

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

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

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

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

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

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

Preprocessor-- #if and #else

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

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

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

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

Preprocessor-- #error and #pragma

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

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

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

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

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

void func() ;

#pragma startup func
#pragma exit func

void main(){
    printf("\nI am in main");
    getch();
}

void func(){
    printf("\nI am in func");
    getch();
}
```


C--Data Segments

To understand the way our C program works, we need to understand the arrangement of the memory assigned to our program.

All the variables, functions, and data structures are allocated memory into a special memory segment known as Data Segment. The data segment is mainly divided into four different parts which are specifically allocated to different types of data defined in our C program.



1. Data Area

It is the permanent memory area. All static and external variables are stored in the data area. The variables which are stored in the data area exist until the program exits.

2. Code Area

It is the memory area which can only be accessed by the function pointers. The size of the code area is fixed.

3. Heap Area

As we know that C supports dynamic memory allocation. C provides the functions like `malloc()` and `calloc()` which are used to allocate the memory dynamically. Therefore, the heap area is used to store the data structures which are created by using dynamic memory allocation. The size of the heap area is variable and depends upon the free space in the memory.

4. Stack Area

Stack area is divided into two parts namely: initialize and non-initialize. Initialize variables are given priority than non-initialize variables.

- All the automatic variables get memory into stack area.
- Constants in c get stored in the stack area.
- All the local variables of the default storage class get stored in the stack area.
- Function parameters and return value get stored in the stack area.
- Stack area is the temporary memory area as the variables stored in the stack area are deleted whenever the program reaches out of scope

C- Memory Management and DMA

The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime.

Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

C- Malloc function

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
    if(ptr==NULL)
    { printf("Sorry! unable to allocate memory");
      exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    { scanf("%d",ptr+i);
      sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

C- Calloc() function

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

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

```
#include<stdlib.h>
```

```
int main(){
```

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

```
    printf("Enter number of elements: ");
```

```
    scanf("%d",&n);
```

```
    ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
```

```
    if(ptr==NULL)
```

```
    {
```

```
        printf("Sorry! unable to allocate memory");
```

```
        exit(0);
```

```
    }
```

```
    printf("Enter elements of array: ");
```

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

```
    {
```

```
        scanf("%d",ptr+i);
```

```
        sum+=*(ptr+i);
```

```
    } printf("Sum=%d",sum);
```

```
    free(ptr);
```

```
    return 0; }
```

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of **realloc()** and **free()** functions

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char name[100];
    char *description;
    strcpy(name, "Nish"); /* allocate memory dynamically */
    description = malloc( 30 * sizeof(char) );
    if( description == NULL )
    {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
        strcpy( description, "Good boy.");
    }
}
```

```
/* suppose you want to store bigger description */
description = realloc( description, 100 * sizeof(char) );
if( description == NULL )
{
    fprintf(stderr, "Error - unable to allocate required
memory\n");
} else
{ strcat( description, "Awosem"); }
printf("Name = %s\n", name );
printf("Description: %s\n", description );
/* release memory using free() function */
free(description);
}
```


C--- Error Handling

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in `<error.h>` header file. So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set **errno** to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current **errno** value.
- The **strerror()** function, which returns a pointer to the textual representation of the current **errno** value.

By using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

C- Error Handling

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;
int main ()
{
    FILE * pf; int errnum;
    pf = fopen ("unexist.txt", "rb");
    if (pf == NULL)
    {
        errnum = errno;
        fprintf(stderr, "Value of errno: %d\n", errno);
        perror("Error printed by perror"); fprintf(stderr,
        "Error opening file: %s\n", strerror( errnum ));
    } else { fclose (pf); } return 0; }
```