C PROGRAMMING

DINESHKUMAR THANGAVEL

PARUL UNIVERSITY

What is C language? Different ways of defining C:

1)C as a mother language

C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the array

strings

functions

file handling

that are being used in many languages like C++

Java

C#

etc.

---

2) C as a system programming language

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

3) C as a procedural language

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

---

4) C as a structured programming language

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

5) C as a mid-level programming language

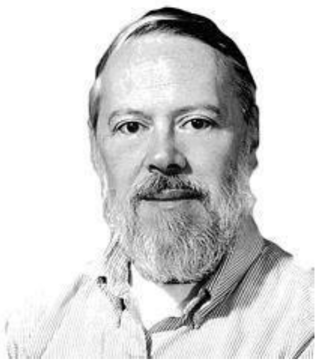
C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

History of C:

Brief History of C Programming Language:



C is developed by **Dennis Ritchie** in the year 1972 at Bell Laboratories(USA). It is a *General-purpose, Structured, Machine Independent, Simple* and *Flexible* programming language. It was mainly developed as a system programming language to write an operating system.

Features of C Programming Language

- **High-Level Language**

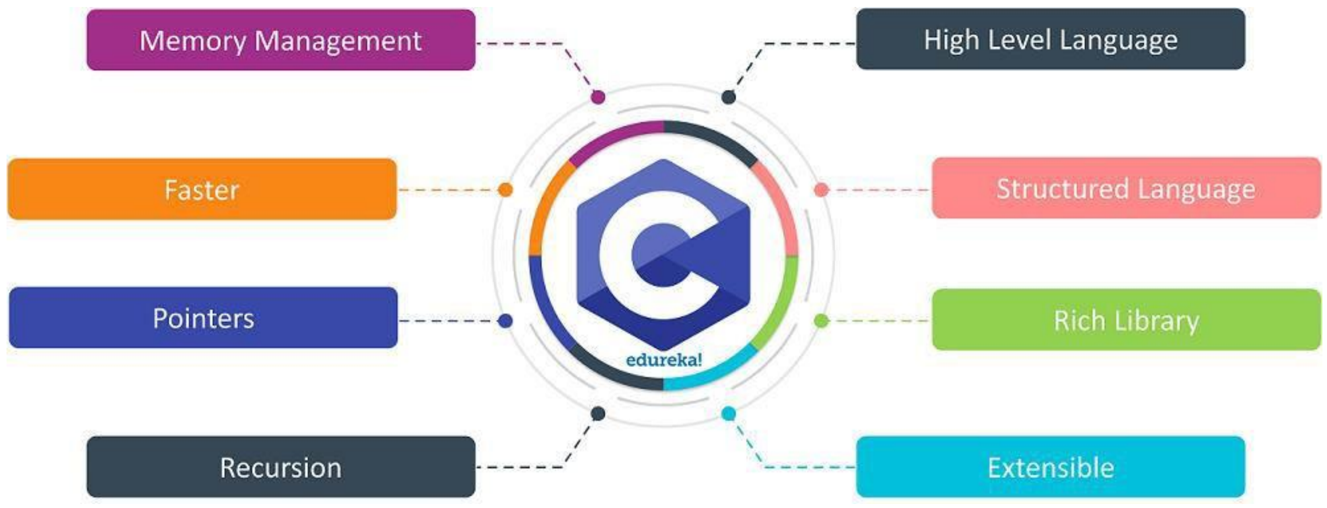
C provides strong **abstraction** in case of its libraries and built-in features that make it machine-independent. It is capable enough to develop system applications such as the kernel, driver, etc.

- **Structured Language**

C Language supports structured programming which includes the use of **functions.** Functions reduce the code complexity and are totally reusable.

- **Rich Library:** Unlike its predecessors, C language incorporates multiple built- in **arithmetic** and **logical** functions along with many built-in libraries which make development faster and convenient.

- **Extensible:** C Language is a High-level language and is also open for **upgrades.** Hence, the programming language is considered to be **extensible** like any other high-level languages.



- **Recursion**

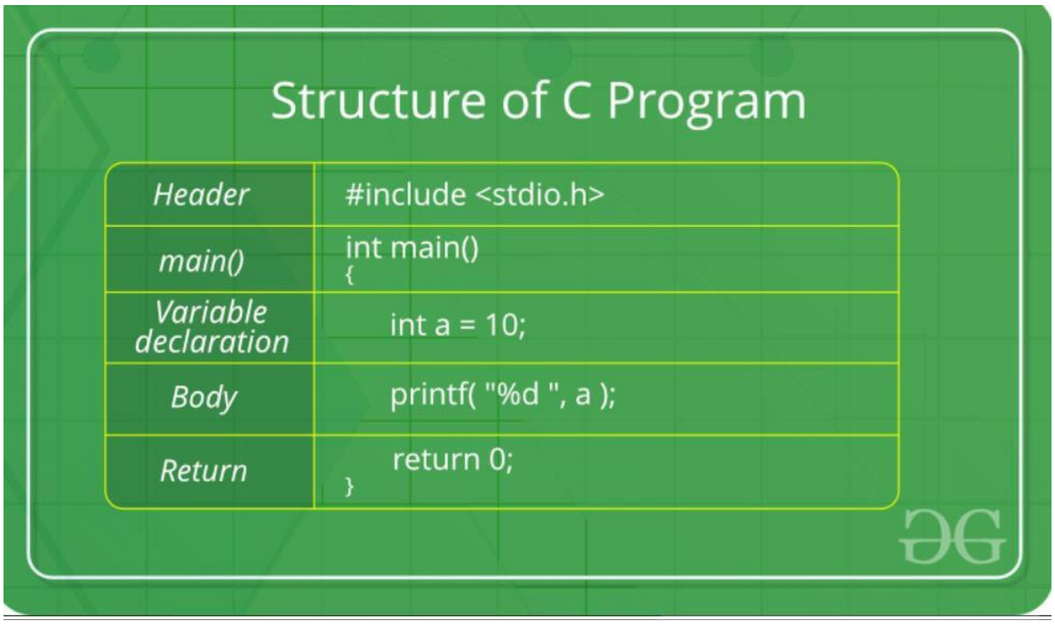
C Language supports function **back-tracking** which involves recursion. In the process of recursion, a function is called within another function for multiple numbers of times.

- **Pointers**

C enables users to directly interact with memory using the pointers. We use pointers in memory, structure, functions, arrays, stack and many more.

- **Faster:** C Language comes with a minimal number of **libraries** and **built-in** functions which makes the compile and execution-times to be less and the system faces low overhead.
- **Memory Management:** C provides the best in class memory management. It can both allocate and deallocate memory dynamically. the **malloc(), calloc(), realloc()** functions are used to allocate memory dynamically and **free()** function is used to deallocate the used memory at any instance of time.

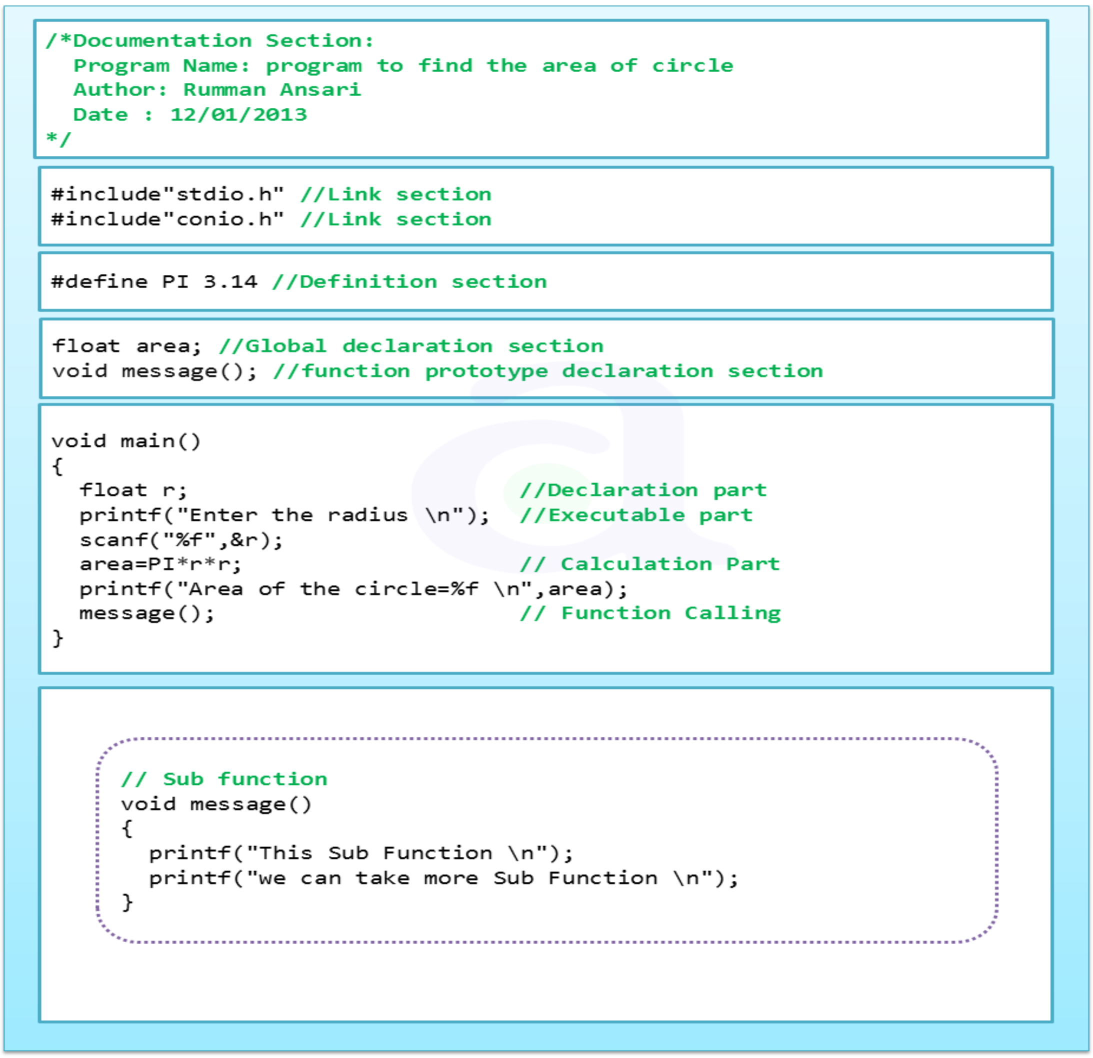
Structure of C programming:



Structure of C Program

Header	#include <stdio.h>
main()	int main() {
Variable declaration	int a = 10;
Body	printf("%d ", a);

Return	return 0;
	}



```
/*Documentation Section:
  Program Name: program to find the area of circle
  Author: Rumman Ansari
  Date : 12/01/2013
*/
```

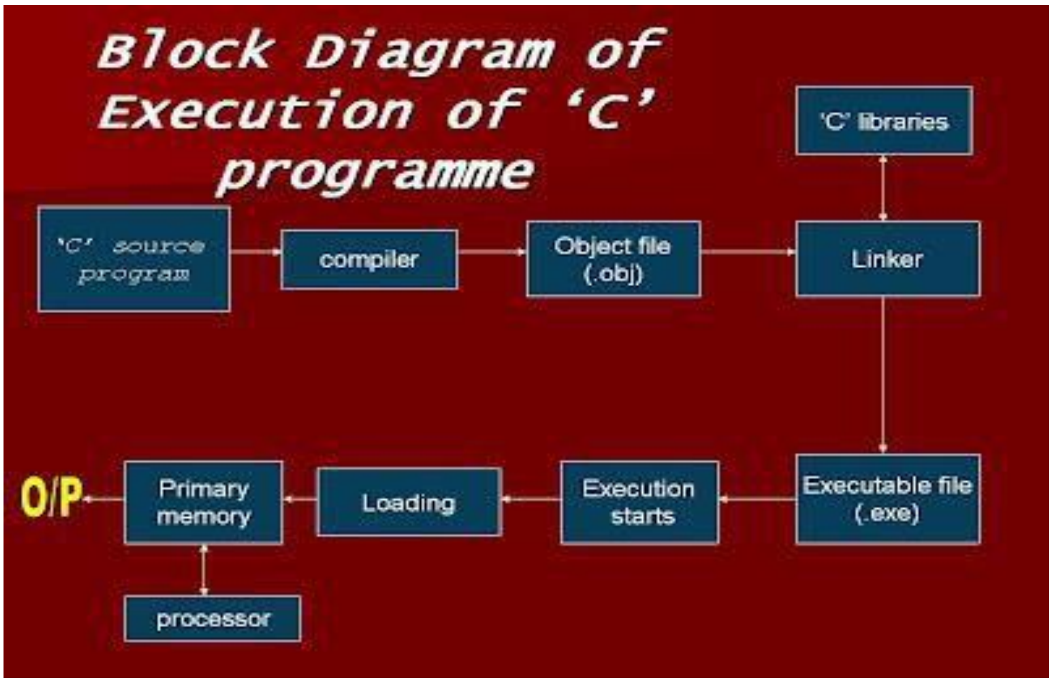
```
#include"stdio.h" //Link section
#include"conio.h" //Link section
```

```
#define PI 3.14 //Definition section
```

```
float area; //Global declaration section
void message(); //function prototype declaration section
```

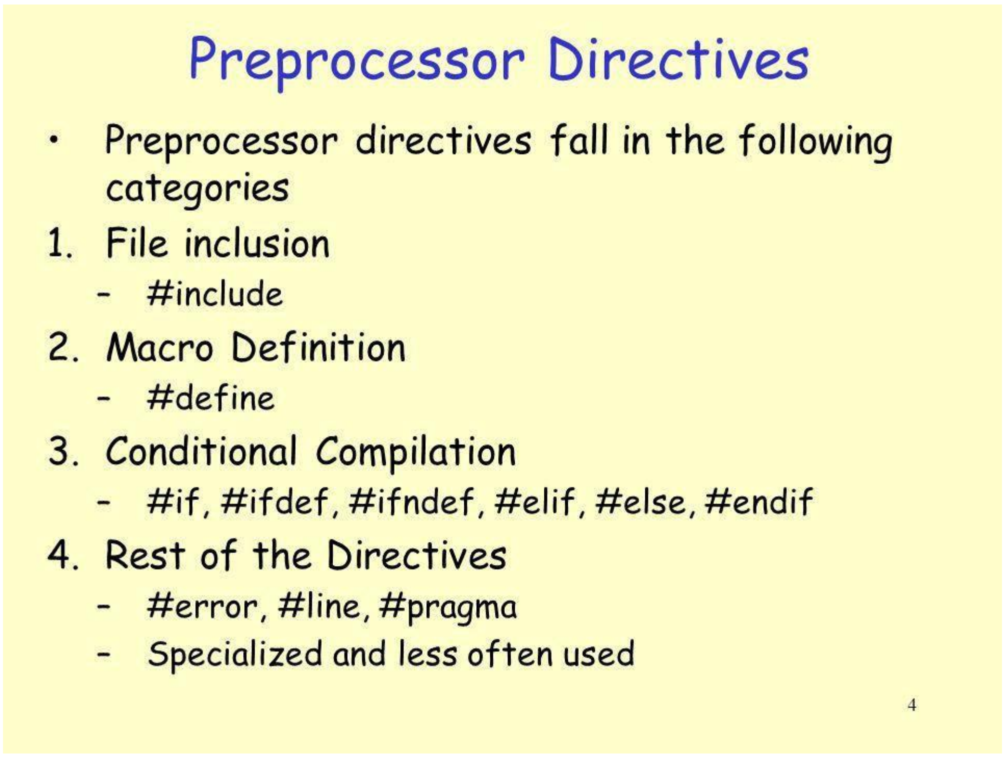
```
void main()
{
  float r;                               //Declaration part
  printf("Enter the radius \n");         //Executable part
  scanf("%f",&r);
  area=PI*r*r;                           // Calculation Part
  printf("Area of the circle=%f \n",area);
  message();                             // Function Calling
}
```

```
// Sub function
void message()
{
  printf("This Sub Function \n");
  printf("we can take more Sub Function \n");
}
```



Pre-processor directives in C:

The pre-processor will **process directives that are inserted into the C source code**. These directives allow additional actions to be taken on the C source code before it is compiled into object code. Directives are not part of the C language itself.



Preprocessor Directives

- Preprocessor directives fall in the following categories
1. File inclusion
   - #include
2. Macro Definition
   - #define
3. Conditional Compilation
   - #if, #ifdef, #ifndef, #elif, #else, #endif
4. Rest of the Directives
   - #error, #line, #pragma
   - Specialized and less often used

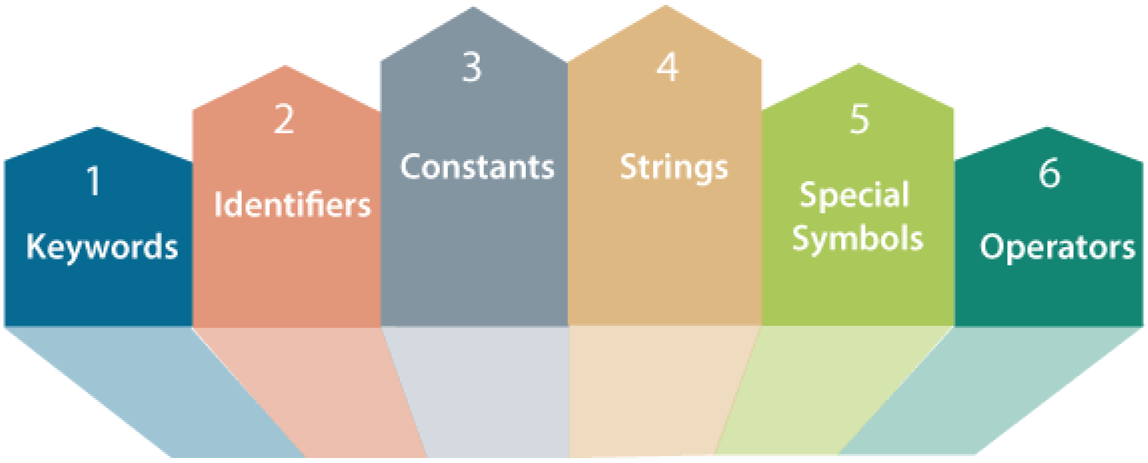
4

Tokens in C

Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For `example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language.

Classification of tokens in C

Tokens in C language can be divided into the following categories:



Classification of C Tokens

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

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

C Identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers: total, sum, average, a12, a_12, ab, place, college etc..

Types of identifiers

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Let's understand through an example.

```
int main()
{
    int a=10;
    int A=20;
    printf("Value of a is : %d",a);
    printf("\nValue of A is :%d",A);
    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.

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

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 −

Show Examples

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 de-numerator.	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

Relational Operators

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

Show Examples

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.

Logical Operators

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

Show Examples

Operator	**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, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

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

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Show Examples

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

Assignment Operators

The following table lists the assignment operators supported by the C language −

Show Examples

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 operator.	C |= 2 is same as C = C | 2

Misc Operators ↦ sizeof & ternary

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

Show Examples

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

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

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

Example:

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

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

    return 0;
}
```

Precedence of Operators in C

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.

Let's understand the precedence by the example given below:

```
1. int value=10+20*10;
```

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

The precedence and associativity of C operators is given below:

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

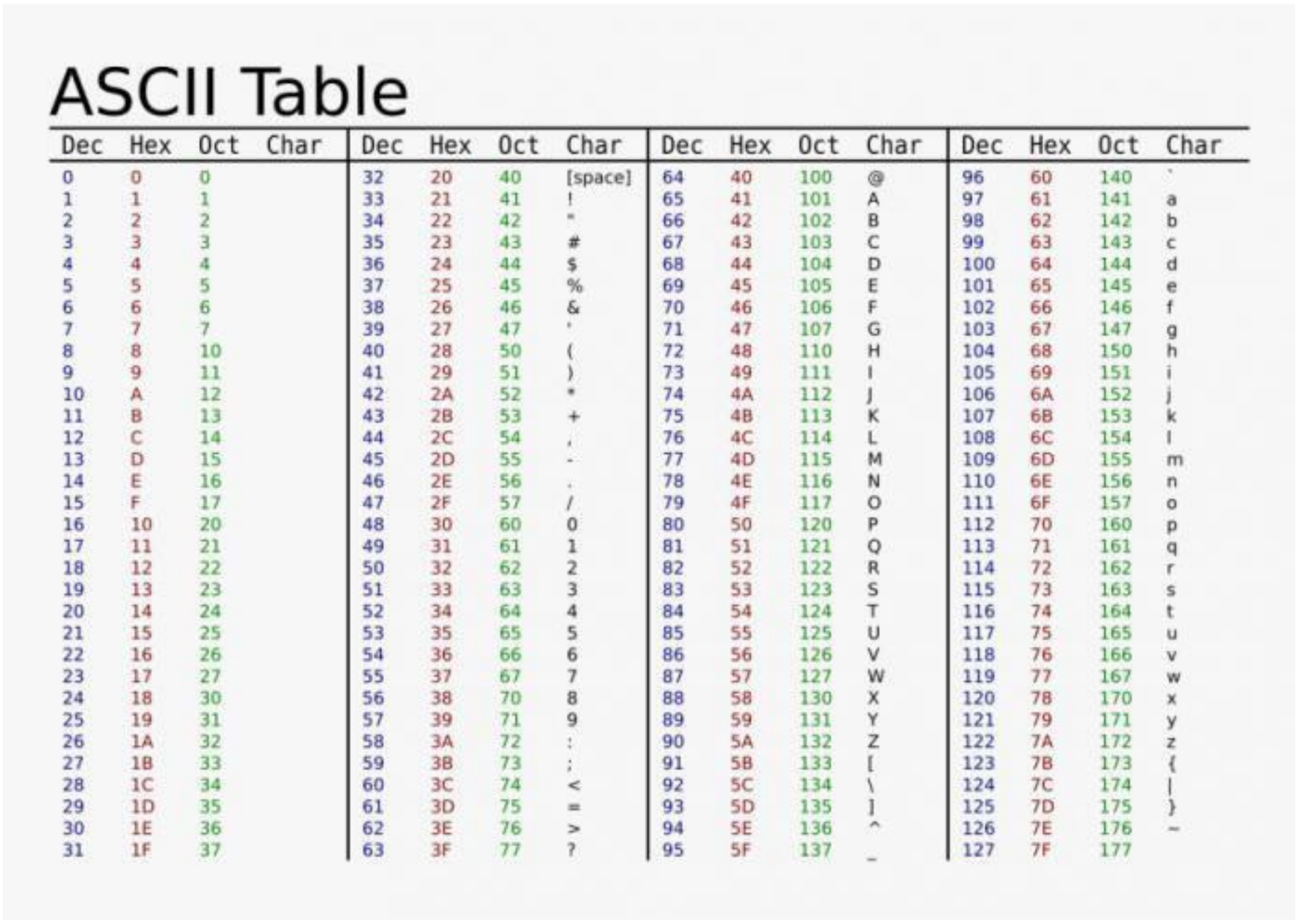
ASCII value in C

What is ASCII code?

The full form of ASCII is the **American Standard Code for information interchange**. It is a character encoding scheme used for electronics communication. Each character or a special character is represented by some ASCII code, and each ascii code occupies 7 bits in memory.

In C programming language, a character variable does not contain a character value itself rather the ascii value of the character variable. The ascii value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127. For example, the ascii value of 'A' is 65.

In the above example, we assign 'A' to the character variable whose ascii value is 65, so 65 will be stored in the character variable rather than 'A'.



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

We will create a program which will display the ascii value of the character variable.

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

```
{
    char ch;   // variable declaration
    printf("Enter a character");
    scanf("%c",&ch); // user input
    printf("\n The ascii value of the ch variable is : %d", ch);
    return 0;
}
```

Escape Sequence in C

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.

List of Escape Sequences in C

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

Example:

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

C Format Specifier

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

The commonly used format specifiers in printf() function are:

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.

%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output.
%p	It is used to print the address in a hexadecimal form.
%c	It is used to print the unsigned character.
%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value.

Type Casting in C

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

Syntax:

```
(type)value;
```

Note: It is always recommended to convert the lower value to higher for avoiding data loss.

Without Type Casting:

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

Static in C

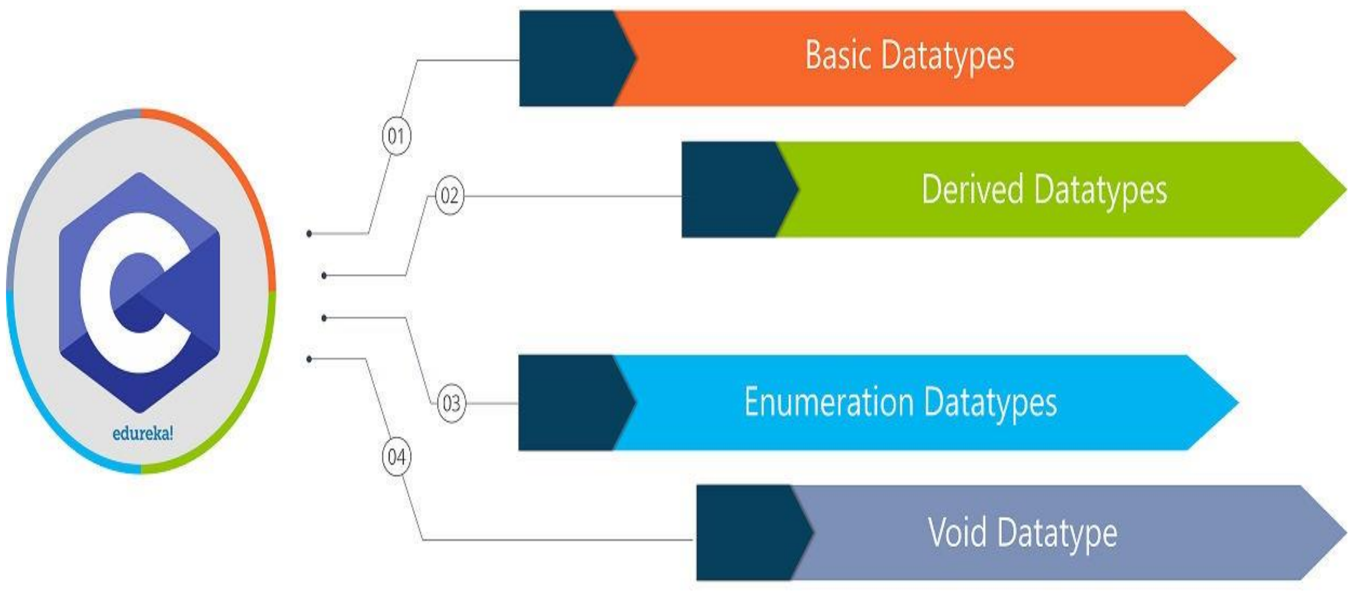
Static is a keyword used in C programming language. It can be used with both variables and functions, i.e., we can declare a static variable and static function as well. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.

Static keyword can be used in the following situations:

- **Staticglobalvariable**
  When a global variable is declared with a static keyword, then it is known as a static global variable. It is declared at the top of the program, and its visibility is throughout the program.
- **Static function**
  When a function is declared with a static keyword known as a static function. Its lifetime is throughout the program.
- **Static local variable**
  When a local variable is declared with a static keyword, then it is known as a static local variable. The memory of a static local variable is valid throughout the program, but the scope of visibility of a variable is the same as the automatic local variables. However, when the function modifies the static local variable during the first function call, then this modified value will be available for the next function call also.

Datatypes and Variables

Datatypes in C are broadly classified into **four** categories as follows.



Basic Datatypes

Basic Datatypes are considered as the most fundamental and primary datatypes that C has to offer. The Datatypes that come under Basic Datatypes are as follows.

Datatype Name	Datatype Size	Datatype Range
short	**1 byte**	-128 to 127
unsigned short	**1 byte**	0 to 255
char	**1 byte**	-128 to 127
unsigned char	**1 byte**	0 to 255
int	**2 bytes**	-32,768 to 32,767
unsigned int	**2 bytes**	0 to 65,535
long	**4 bytes**	-2,147,483,648 to 2,147,483,647
unsigned long	**4 bytes**	0 to 4,294,967,295
float	**4 bytes**	3.4E-38 to 3.4E+38

double	**8 bytes**	1.7E-308 to 1.7E+308
long double	**10 bytes**	3.4E-4932 to 1.1E+4932

Derived Datatypes

A **derived type** is formed by using one or more basic **types** in combination. They are the object types whose functionalities are predefined in the C libraries.

- Function types
- Pointer types
- Array types
- Structure types
- Union types

Enumeration Datatypes

Enumerated Datatypes are used to declare **Integral constants** in C programming language so that the integral constant names are easy to remember and maintain. The keyword **enum** is used to declare enumerated datatypes.

example: **enum** plug{on = 1, off = 0};

Void Datatypes

The void data type is an empty data type that is used as a return type for the functions that return no value in C.

example:

void function(int n)
int function(void)

Variables in C

Variable is defined as the reserved memory space which stores a value of a definite datatype. The value of the Variable is not constant, instead, it allows changes. There are mainly five types of variables supported in C.

- local variable
- global variable
- **static variable**
- automatic variable
- external variable

Local Variables

Any variable that is declared at the inside a code block or a function and has the scope confined to that particular block of code or function is called to be a **local variable.**

//Example

```
Void Edu()
{
        int Local_variable=10;
}
```

Global Variables

Any variable that is declared at the outside a code block or a function and has the scope across the entire program and allows any function to change it's value is known as **Global Variable**.

//Example

```
int Global_variable=10;
void Edu()
{
       int Local_variable=20;
}
```

Static Variables

Any variable that is declared using the keyword **static** is known as a **Static Variable**. Static Variables retain the declared value throughout the entire execution of the program and will not be changed between multiple function calls.

//Example

```
void Edu()
{
     int Local_variable =10;//
     static int Static_variable =10;
     Local_variable=Local_variable+1;
     Static_variable =Static_variable+1;
     printf("%d,%d",x,y);
}
```

Automatic Variables

Automatic Variables can be declared by using the keyword **auto.** By default, all the variables declared in C language are Automatic Variables.

//Example

```
void main()
{
      int Local_variable=10;  //(automatic default)
      auto int auto=20;   //(automatic variable)
};
```

External Variables

External Variables are declared by using the **extern** keyword. We can share a variable in multiple C source files by using an external variable.

//Example

```
extern external=10;
```

Formatted input and output : printf() and scanf() in C

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).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

```
printf("format string",argument_list);
```

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

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

```
scanf("format string",argument_list);
```

Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

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

Output

```
enter a  number:5
cube of number is:125
```

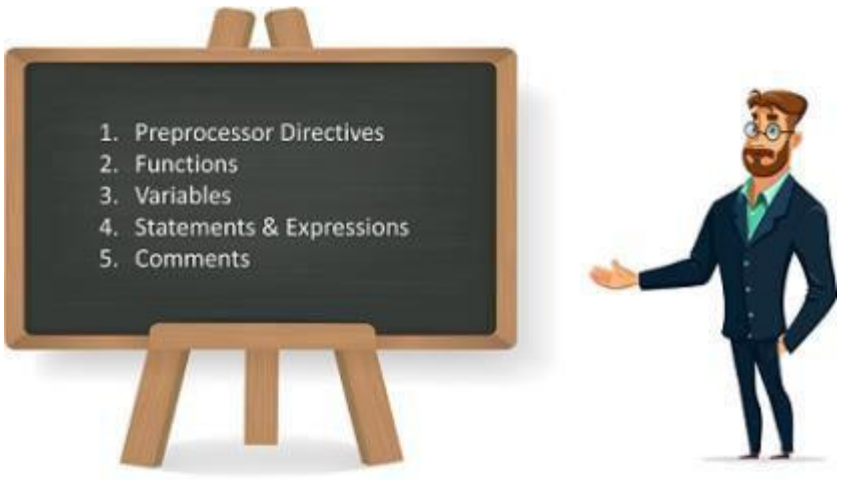
The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number*number*number)** statement prints the cube of number on the console

Let us execute our first C Program.

First C Program

In this C Programming Tutorial, we will understand the **basic structure of a C Program**. Any basic C program consists of the following sections



- **Preprocessor Directives**

Preprocessor Directives are declared at the beginning of every C Program with a special character #. They are defined as predefined **functions** or **Macro** programs that are invoked and compiled by the C Compiler during the **compilation** process.

- **Functions**

Functions in C Program are defined as the **subdivided program** of the main program. Functions provide code *reusability* and decrease *code complexity.*

- **Variables**

Variable is defined as a name declared to a store the values in a C Program. Every variable used in C has a specific data type that determines the *size* and *layout* of the variable's memory.

- **Statements & Expressions:**Statements are nothing but the instructions we provide to the computer to compile and the expressions, on the other hand, are considered to be the *mathematical* or the *logical* statements which yield into a resultant value.

- **Comments**

Comments are the messages written by the developer in the code for easy understanding of the logic of the code used while programming. The comments will not be compiled by the compiler. the comments are written within ***//*** or ***/* */.***

Let us now execute our first Hello World Program.

```
#include<stdio.h>
//main fn
int main(){
/* printing hello world*/
```

```
    printf("Hello World");
    return 0;
}
```

//Output

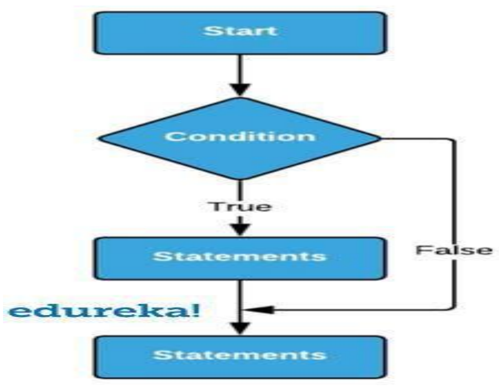
```
Hello World
```

Conditional/ Decision making Statements in C Programming:

Conditional Statements in C language can be defined as the programming statements that are designed to decide the execution flow of the statements of the program over the specified mathematical or logical condition. The important **conditional statements** in this C Programming tutorial are as follows.

if

If Statement in C language is a programming conditional statement that executes a code segment over a condition, provided if it is true and valid. Below is the flowchart for **if** condition.



//Example

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

[redacted]

//Output

```
Enter a number:4
your number is 4 and it is an even number
```

else-if

Else If Conditional Statement in C language is used to execute one statement out of the two. The Conditional Statement executes the code segment provided is *true and valid*. Below is the flowchart for the **else-if** condition.

//Example

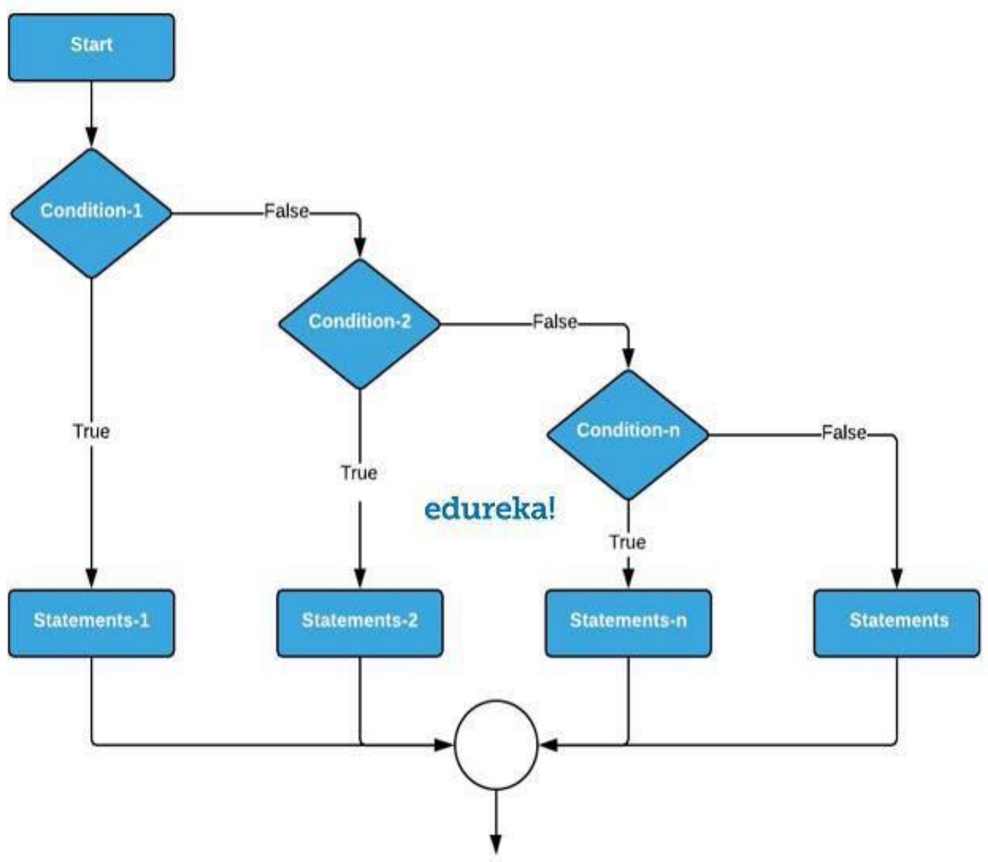
```
#include<stdio.h>
int main()
{
    int number=0;
    printf("enter a number:");
    scanf("%d",&number);
    if(number%2==0)
    {
        printf("your number is %d and it is an even
number",number);
    }
    else
    {
        printf("your number is %d and it is an odd
number",number);
    }
 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

//Output

```
Enter a number:5
your number is 5 and it is an odd number
```

else-if ladder

Else if Ladder in C language is set of consecutive Else-If statements which are used to *execute one true and valid statement* out of a collection of given statements. Below is the flowchart for the **else-if ladder** condition.



```
#include<stdio.h>
int main()
{
        int number=0;
        printf("enter a number:");
        scanf("%d",&number);
        if(number==10)
        {
                printf("your inputted number is equals to 10");
        }
```

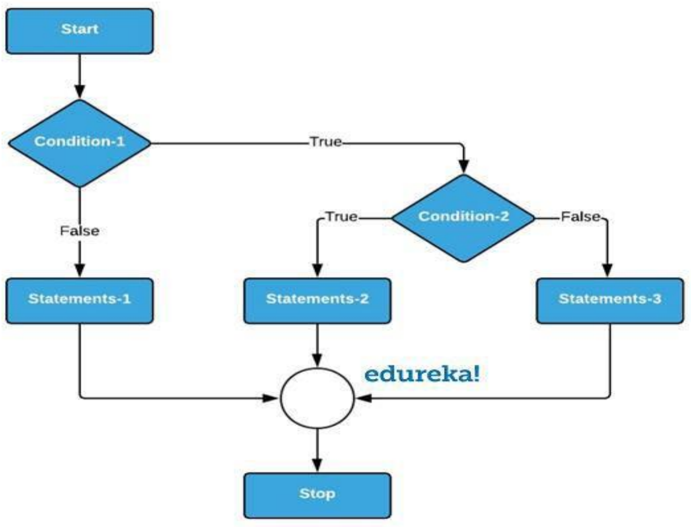
```
     else if(number==50)
    {
            printf("your inputted number is equal to 50");
    }
    else if(number==100)
    {
            printf("your inputted number is equal to 100");
    }
    else
   {
            printf("your inputted number is not equal to 10, 50 or 100");
   }
   return 0;
}
```

//Output

```
enter a number:5
your inputted number is not equal to 10, 50 or 100
```

Nested if

Nested-If in C language is a conditional statement where one Else-If statement is embedded into another If statement. Below is the flowchart for the **nested-if** condition.



//Example

```
#include<stdio.h>
int main()
{
        int v1, v2;
        printf("Enter the value of First variable :");
        scanf("%d",&v1);
        printf("Enter the value of Second variable :");
        scanf("%d",&v2);
        if (v1 != v2)
        {
                printf("First variable is not equal to Second variablen");
                if (v1<v2)
                {
                        printf("First variable is not equal to Second
variablen");
                }
                else
                {
                        printf("Second variable is greater than First
variablen");
                 }
         }
         else
        {
                 printf("First variable is equal to Second variablen");
        }
        return 0;
}
```

//Output:

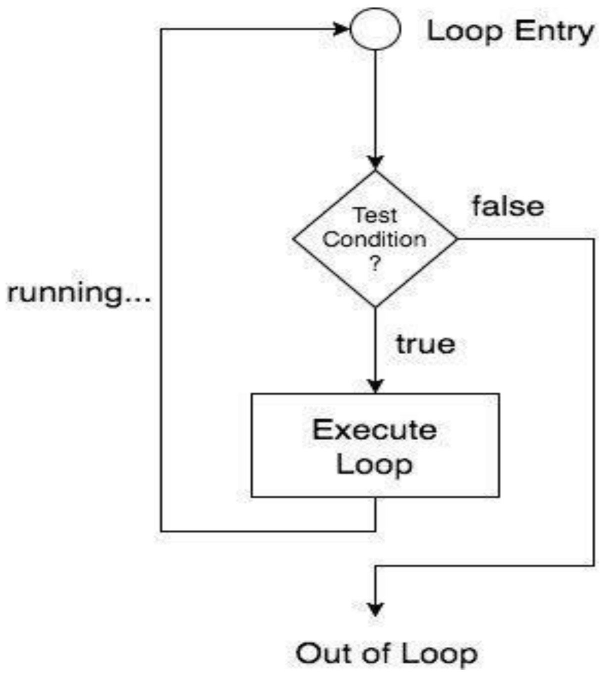
```
Enter the value of First variable :12
Enter the value of Second variable :21
First variable is not equal to Second variable
Second variable is greater than First variable
```

Looping Statements:

Loops in C are defined as a programming statement that is designed to execute a particular code segment for a *specific number* of times or until a particular *condition* provided is satisfied. There are mainly three-loop statements available in C.

How it Works:

The below diagram depicts a loop execution,



As per the above diagram, if the Test Condition is true, then the loop is executed, and if it is false then the execution breaks out of the loop. After the loop is successfully executed the execution again starts from the Loop entry and again checks for the Test condition, and this keeps on repeating.

The sequence of statements to be executed is kept inside the curly braces { } known as the **Loop body**. After every execution of the loop body, **condition** is verified, and if it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed, and execution breaks out of the loop.

Types of Loop

There are 3 types of Loop in C language, namely:

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

---

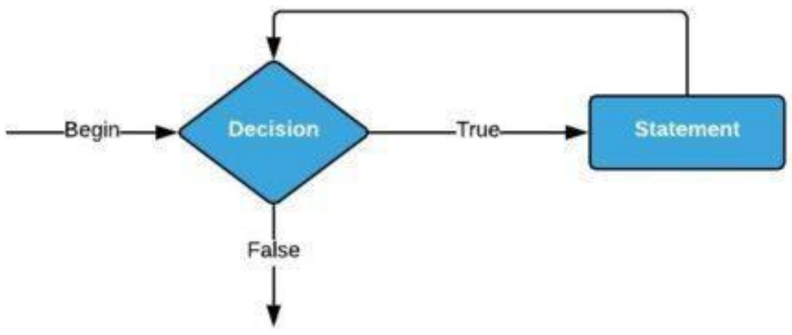
while loop

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

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

Syntax :

```
variable initialization;

while(condition)

{

    statements;

    variable increment or decrement;

}
```



Example: Program to print first 10 natural numbers

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

```
void main( )
{
    int x;
    x = 1;
    while(x <= 10)
    {
        printf("%d\t", x);
        /* below statement means, do x = x+1, increment x
by 1*/
        x++;
    }
}
```

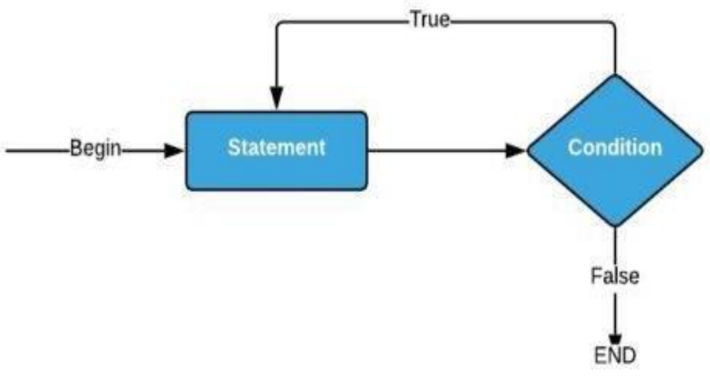
```
1 2 3 4 5 6 7 8 9 10
```

---

do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be **false**. General syntax is,

```
do

{

    .....

    .....

}

while(condition)
```



example: Program to print first 10 multiples of 5.

```
#include<stdio.h>



void main()

{

    int a, i;

    a = 5;

    i = 1;

    do
    {
        printf("%d\t", a*i);
        i++;
    }
    while(i <= 10);
}
```

while loop vs do-while loop in C

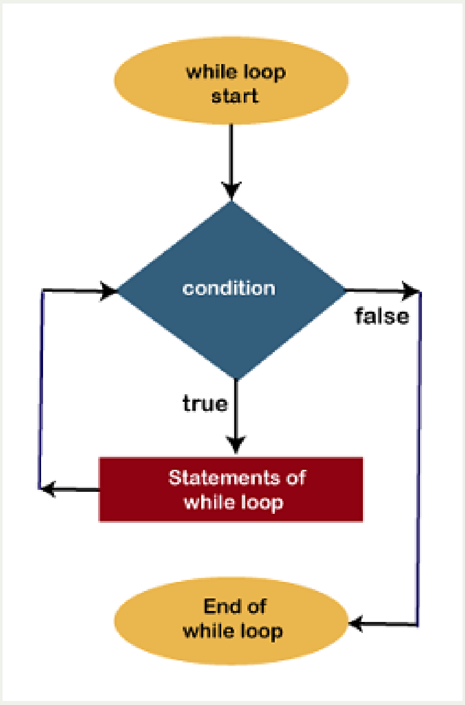
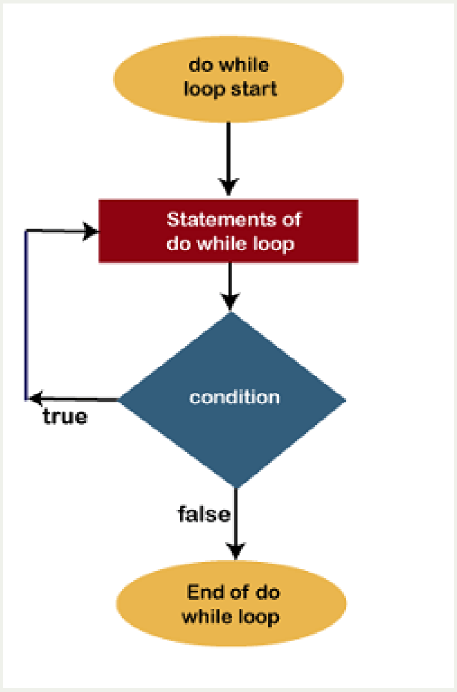
Let's see some Comparison between the while loop and do-while loop in the C language

SR.NO	while loop	do-while loop
1.	While the loop is an entry control loop because firstly, the condition is checked, then the loop's body is executed.	The do-while loop is an exit control loop because in this, first of all, the body of the loop is executed then the condition is checked true or false.
2.	The statement of while loop may not be executed at all.	The statement of the do-while loop must be executed at least once.
3.	The while loop terminates when the condition becomes false.	As long as the condition is true, the compiler keeps executing the loop in the do-while loop.
4.	In a while loop, the test condition variable must be initialized first to check the test condition in the loop.	In a do-while loop, the variable of test condition Initialized in the loop also.
5.	In a while loop, at the end of the condition, there is no semicolon. **Syntax:** `while (condition)`	In this, at the end of the condition, there is a semicolon. **Syntax:** `while (condition);`

6.	While loop is not used for creating menu-driven programs.	It is mostly used for creating menu-driven programs because at least one time; the loop is executed whether the condition is true or false.
7.	In a while loop, the number of executions depends on the condition defined in the while block.	In a do-while loop, irrespective of the condition mentioned, a minimum of 1 execution occurs.

8.	**Syntax of while loop:**
	`while (condition)`
	`{`
	`Block of statements;`
	`}`
	`Statement-x;`
		Syntax of do-while loop:
		`do`
		`{`
		`statements;`
		`}`
		`while (condition);`
		`Statement-x;`

9.	**Program of while loop:**
	`Program of while loop:`
	
	`#include`
	`#include`
	`Void main()`
	`{`
	`int i;`
	`clrscr();`
	`i = 1;`
	`while(i<=10)`
	`{`
	`printf("hello");`
	`i = i + 1;`
	`}`
	`getch();`
	`}`
		Program of do-while loop:
		`#include`
		`#include`
		`Void main()`
		`{`
		`int i;`
		`clrscr();`
		`i = 1;`
		`do`
		`{`
		`printf("hello");`
		`i = i + 1;`
		`}`
		`while(i<=10);`
		`getch();`
		`}`

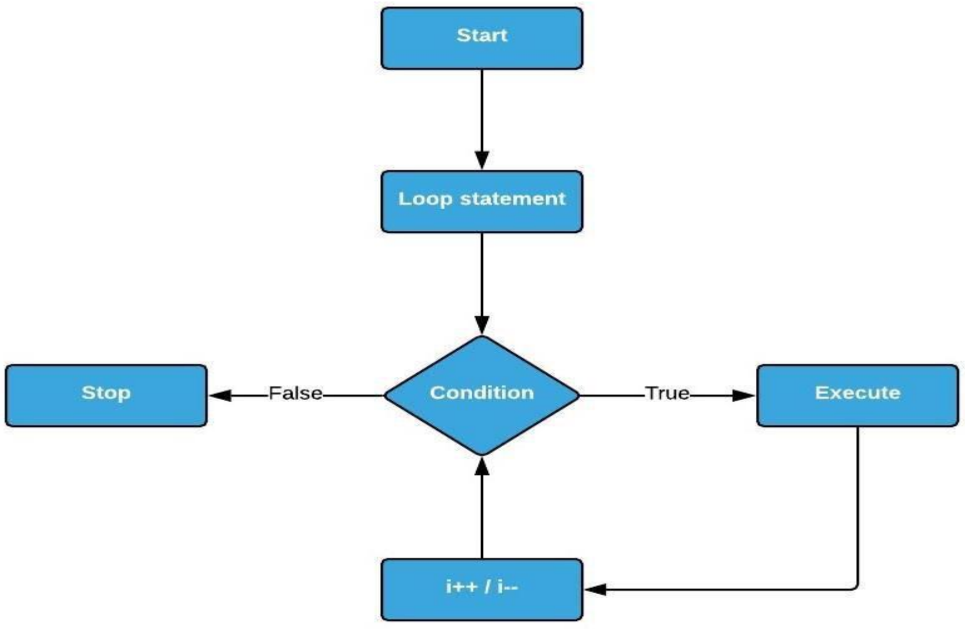
10.	**Flowchart of while loop:**  while loop start condition false true Statements of while loop End of while loop	**Flowchart of do-while loop:**  do while loop start Statements of do while loop condition true false End of do while loop

for loop

for loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop.**. General format is,

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

{

    statement-block;

}
```

In for loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.



The for loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

Example: Program to print first 10 natural numbers

```
#include<stdio.h>


void main( )

{

    int x;
    for(x = 1; x <= 10; x++)

    {

        printf("%d\t", x);

    }

}
```

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

Nested for loop

We can also have nested for loops, i.e one for loop inside another for loop. Basic syntax is,

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

{

    for(initialization; condition; increment/decrement)

    {

        statement ;

    }

}
```

Example: Program to print half Pyramid of numbers

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

```
void main( )
{
    int i, j;
    /* first for loop */
    for(i = 1; i < 5; i++)
    {
        printf("\n");
        /* second for loop inside the first */
        for(j = i; j > 0; j--)
        {
            printf("%d", j);
        }
    }
}
```

Copy

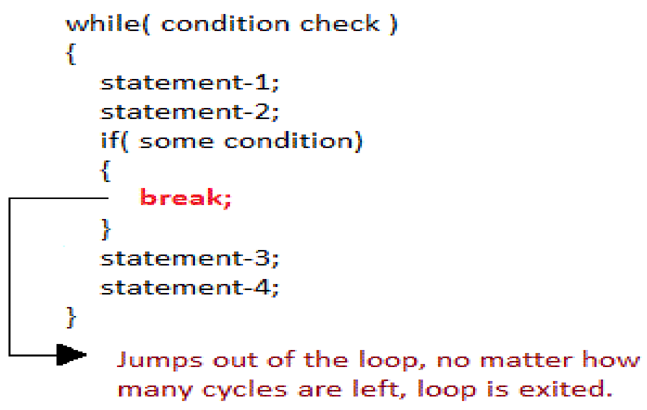
```
1
21
321
4321
54321
```

Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes **true**. This is known as jumping out of loop.

1) break statement

When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

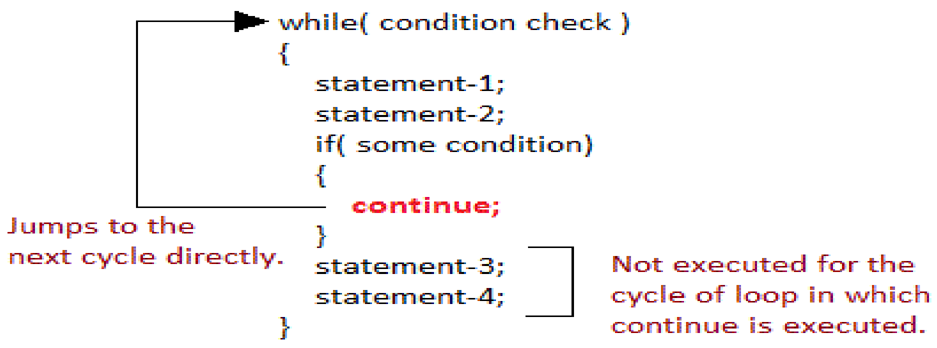


```
while( condition check )
{
    statement-1;
    statement-2;
    if( some condition)
    {
        break;
    }
    statement-3;
    statement-4;
}
```

Jumps out of the loop, no matter how many cycles are left, loop is exited.

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.



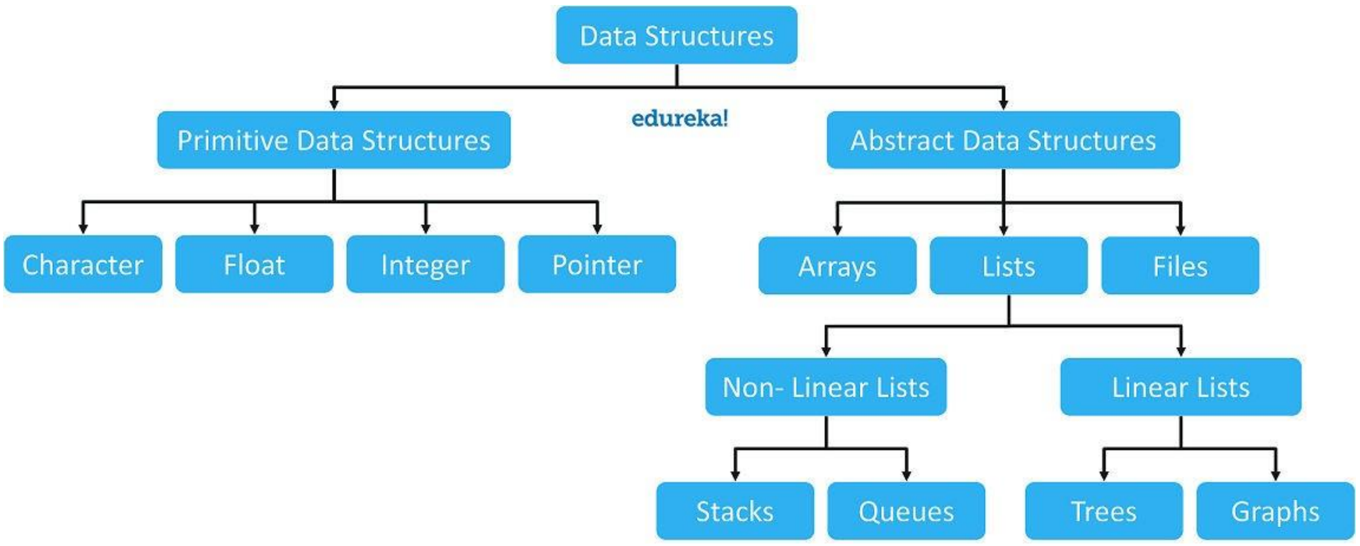
Example:

```
#include<stdio.h>

int main()
{
    int i;

    for(i = 0; i < 10; i++)
    {
        if( i % 4 == 0 )
        {
            /*
                when i is divisible by 4
                continue to the next iteration
            */
            continue;
        }
        printf("%d\n", i);
    }

    // signal to operating system everything works fine
    return 0;
}
```

Data Structures in C Programming

A **Data structure** can be defined as a collection of **data** values, the relationships among them, and the functions that are applied on to the **data**. They are broadly classified as follows.

- Primitive Data Structures / Built-in Data Structures
- Abstract Data Structures / User-Defined Data Structures



Arrays

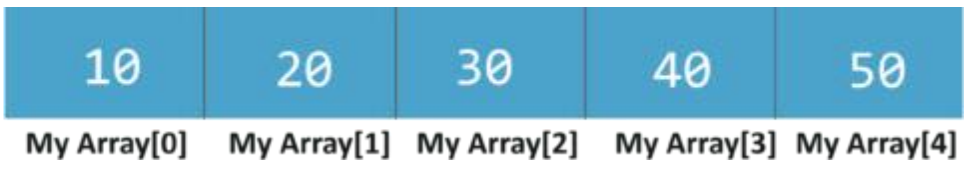
An **array** is defined as the collection of similar type of data items stored at contiguous memory locations. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

There are three different types of Arrays, namely:

- One-Dimensional Array
- Two-Dimensional Array
- Multi-Dimensional Array

One-Dimensional Array

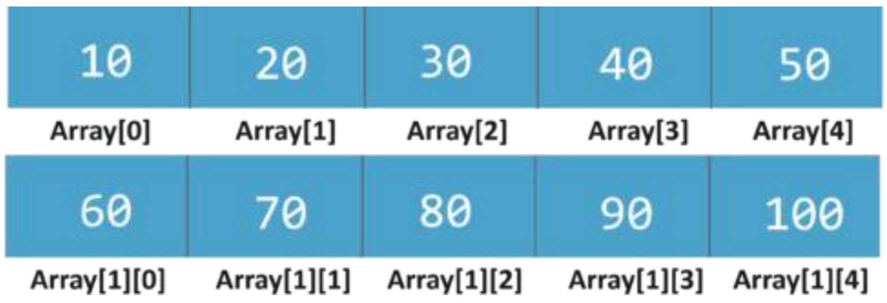
The One-dimensional array can be defined as an array with a single row and multiple columns. The elements in the 1D array are accessed using their index numbers.



```
int arrayone[10];
```

Two-Dimensional Array

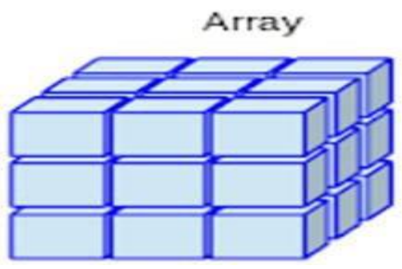
The two-dimensional array can be defined as an array of arrays. The **2D** array is organized as matrices which can be represented as the collection of rows and columns. The elements of the array are accessed using their intersection coordinates.



```
int arraytwo[10][5];
```

Multi-Dimensional Array

The multi-dimensional array can be defined as an array of arrays. The 3D array is organized as 3D matrix which can be represented as the collection of multiple rows and columns. The elements of the array are accessed using their 3D intersection coordinates.



```
int arraythree[10][10][10];
```

Why do we need it? In this article, you will learn about the data structure array and its types and why is it a widely used data structure. Let's get into it.

An Array is a sequential collection of elements, of the same data type. They are stored sequentially in memory. An Array is a data structure that holds a similar type of elements. The array elements are not treated as objects in c like they are in java.

Imagine you are at a musical instrument store and I tell you to arrange all the keyboards under the brand Casio at one place one above the other. This sequential collection of records is called an Array. An array is a sequential collection of elements of the same data type. In our example above, Casio is the data type and all the keyboards you collected are of the brand Casio. All the elements in an array are addressed by a common name.

There are two types of arrays:

1. Single dimensional array
2. Array Initialization
3. Accessing Array at any Point
4. Multidimensional array

SINGLE DIMENSIONAL ARRAY

The syntax for declaring a *Single Dimensional Array:*

```
Datatype arrayName[arraySize];
```

We have a data type that can be any of the basic data types like int, float or double. Array Name is the name of the array and we declare the size of the array.

In our above example, the array will be,

```
Casio keyboardArray[100];
```

Let's consider another example:

```
int test[30];
```

The array test will hold the elements of type int and will have a size 30.

ARRAY SIZE:

Array size is given at the time of declaration of the array. Once the size of the array is given it cannot be changed. The compiler then allocates that much memory space to the array.

Consider the Example:

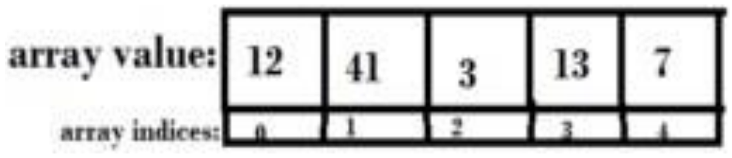
```
int test[20];
```

In the example above, we have an array test, of type int. We have given the array size to be 20. This means that 20 consecutive memory locations will be left free for the array in the memory.

ARRAY INDEX AND INITIALIZATION

A number associated with each position in an array and this number is called the *array index*. Its starts from 0 and to the last element, that is the size of the array minus one. The minus one is there because we start counting from zero and not one. Array indices always begin from zero.

Consider this example, this is the age array.



Here the array contains the values 12,41,3,13,7 and the indices are 0,1,2,3,4. If we want to represent an element at index 4 it is represented as age[4] and the value 7 will be displayed.

By default, the array contains all zero values.

Array initialization is done at the time of declaration. This can also be carried out later if the user enters the array value as and when needed.

INITIALIZATION DURING DECLARATION:

An array can be initialized during declaration. This is done by specifying the array elements at the time of declaration. Here the array size is also fixed and it is decided by us.

Consider the code,

```
#include<stdio.h>

int main()

{

int arr[] = { 10, 20, 30, 40 } ;

return 0;  }
```

EXPLANATION:

In the above example, we create an array of type int and with the name arr. We directly specify the array elements. The size of the array is decided by counting the number of elements in our array. In this case, the size is 4.

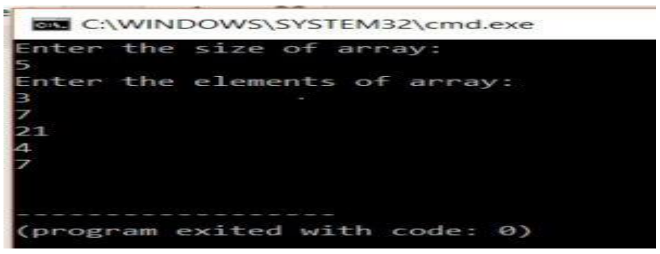
INITIALIZATION BY USER:

In this method, we let the user decide the size of the array. In this case, we need a variable to hold the size of the array and a for loop to accept the elements of the array. We assign a random size at the time of declaration and use only as needed. The size at the start is usually at the higher side. We have a variable i to control the for loop.

Consider the example,

```
#include<stdio.h>

int main()

{

int arr[50],n,i ;

printf("Enter the size of array:n");

scanf(“%d ”,n);

printf("Enter the elements of array:n");

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

{

            Scanf(“%d ”,arr[i]);

}



return 0;

}
```

OUTPUT:



```
C:\WINDOWS\SYSTEM32\cmd.exe
Enter the size of array:
5
Enter the elements of array:
3
7
21
4
7

------------------
(program exited with code: 0)
```

EXPLANATION:

In the above program, we declare an array of size 50. We then ask the user to enter the number of elements he wishes to enter in this array. We then accept the array elements entered by the user.

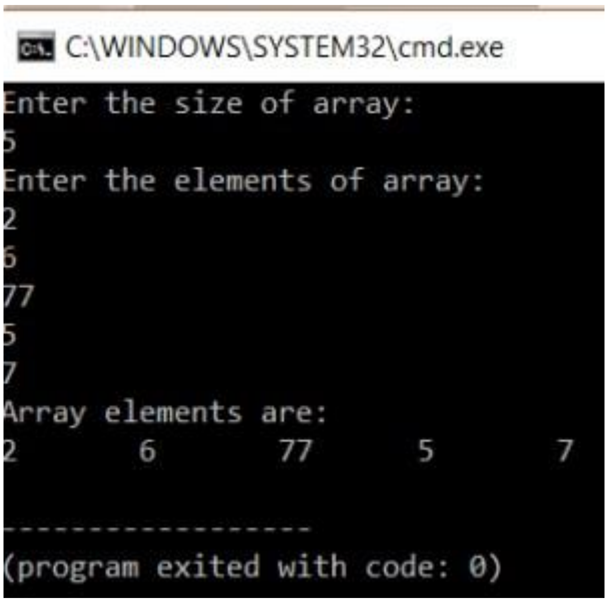
DISPLAYING THE ARRAY:

Displaying the array also requires the for-loop. We traverse to the entire array and display the elements of the array.

Here is an example,

```
#include<stdio.h>

int main()

{

int arr[50],n,i ;

printf("Enter the size of array:n");

scanf(“%d ”,n);

printf("Enter the elements of array:n");

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

{

            Scanf(“%d ”,arr[i]);

}

printf("Array Elements are:n");

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

{

            Printf(“%d  ”,arr[i]);

}

return 0;

}
```

OUTPUT:



```
Enter the size of array:
5
Enter the elements of array:
2
6
77
5
7
Array elements are:
2        6        77       5        7

------------------
(program exited with code: 0)
```

EXPLANATION:

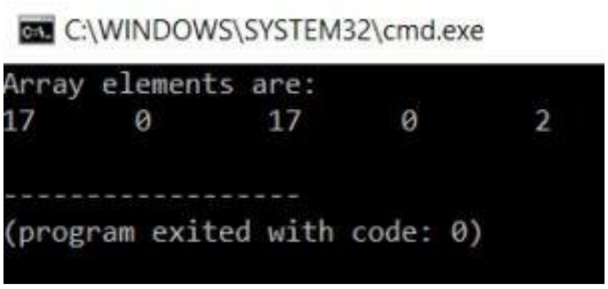
In the above program, we declare an array of size 50. We then ask the user to enter the number of elements he wishes to enter in this array. We then accept the array elements entered by the user. We then use a for loop again to display the array elements.

ACCESSING THE ARRAY AT ANY POINT:

Accessing array elements is simple and is done by using the array index. Have a look at the code below.

```
#include<stdio.h>

int main()

{

int arr[5],i ;

arr[4]=2;

arr[2]=17;

arr[0]=17;

printf("Array elements are: n");

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

{

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

}

return 0;
}
```

OUTPUT:



```
C:\WINDOWS\SYSTEM32\cmd.exe
Array elements are:
17      0       17      0       2

------------------
(program exited with code: 0)
```

EXPLANATION:

In the program above, we have an array of size 5. We enter elements at different locations using array index. We print the array to get the above output.

By Default, all the array elements are zero. What happens if we cross the array size?

In c, if we try to access the elements out of bound, error may not be shown by the compiler but we will not get proper output.

MULTI-DIMENSIONAL ARRAY:

Arrays of arrays are multi-dimensional arrays. This is because each element in a multi-dimensional array has an array of its own. We need n for loops to iterate through a multidimensional array depending on the dimensions.

SYNTAX FOR DECLARING MULTI_DIMENSIONAL ARRAYS:

```
Datatype arrayname[size1][size2].....[size n];
```

Consider the example,

```
int a[10][20];
```

The size of the above array will be 10*20 that is 200 elements.

Similarly, we can have two or three or even more dimensional arrays.

Each dimension requires one for loop. So, the two-dimensional array requires two- and three-dimensional array requires three.

Consider the code:

```
#include<stdio.h>

int main()

{
        int arr[3][2] = {{0,1},{2,3}, {4,5}};



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

        {

                for (int j = 0; j < 2; j++)

                {

                        printf( "Element:  %d n”,arr[i][j]);

                }

        }


        return 0;

}
```

EXPLANATION:

In the above code, we display a 3*2 matrix. This array has 3 rows and 2 columns. We have 2 for loops. Each responsible for one dimension of the array. The outer for loop takes care of rows and inner of columns.

Similarly, we can write a code for three-dimensional array and there will be three for loops and each dimension will be controlled by one for loop.

Strings

The string is defined as a one-dimensional array of characters terminated by a null character. The character array or the string is used to store text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0.

Two types of strings are:

- char array
- string literal

The String functions available in C language are as follows

Function	Functionality
strlen()	Returns the length of the String
strcpy()	Copies String from Source to Destination
strcat()	Joins two Strings and stores result in first
strcmp()	Compares the given two Strings
strrev()	Reverses the String
strlwr()	Converts String to Lower case
strupr()	Converts String to Upper case
strchr()	String scanning function
strncat()	Concatenates String with the part of another
srtncmp()	Compares the parts of two Strings
strncpy()	Copies a part of the String

//Example

```
#include<stdio.h>
#include<string.h>
int main( )
{
      int len;
      char text[10]="saveetha" ;
      length = strlen(array) ;
      printf ( "string length is = %d n" , length ) ;
      return 0;
}
```

//Output

```
string length is = 7
```

Functions

A **Function** can be defined as a subdivided program of the main program enclosed within flower brackets. Functions can be called by the main program to implement its functionality. This procedure provides Code Re-usability and Modularity.

Two types of Functions are:

- Library Functions
- User-Defined Functions

Advantages of Functions

- Writing the same code repeatedly in a program will be avoided.
- Functions can be called any number of times in a program.
- Tracking a C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.

Rules to be followed to use functions:

- **Function Declaration**

A function is required to declared as **Global** and the **name** of the function, **parameters** of the function and **return type** of the function are required to be clearly specified.

- **Function Call**

While calling the function from anywhere in the program, care should be taken that the **datatype** in the argument list and the **number of elements** in the argument list are matching.

- **Function Definition**

After the function is **declared,** it is important that the function includes **parameters declared, code segment,** and **the return value.**

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

- **Four Aspects of using Functions in C Programming**
  - Function **without arguments** and **without return value**
  - Function **without arguments** and **with the return value**
  - Function **with arguments** and **without return value**
  - Function **with arguments** and **with the return value**

Example for Function without argument and return value

Example 1

```
#include<stdio.h>

void printName();

void main ()
{
    printf("Hello ");
   printName();
}
void printName()
{
   printf("Javatpoint");
}
```

Output

```
Hello Javatpoint
```

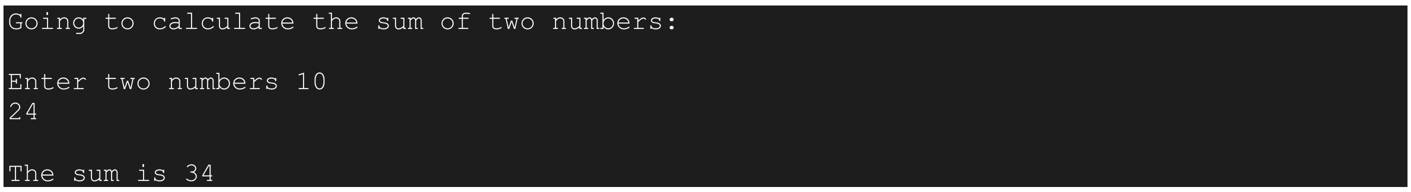
Example for Function without argument and with return value

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}
intsum()
{


    inta,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Example for Function with argument and without return value

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

Output



```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example for Function with argument and with return value:

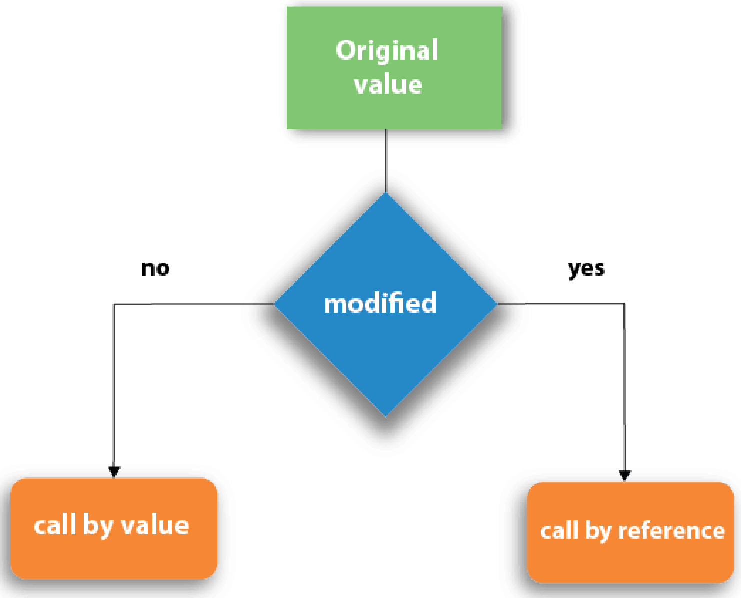
```
#include<stdio.h>
int even_odd(int);
void main()
{
 int n,flag=0;
 printf("\nGoing to check whether a number is even or odd");
 printf("\nEnter the number: ");
 scanf("%d",&n);
 flag =even_odd(n);
 if(flag == 0)
 {
    printf("\nThe number is odd");
  }
  else
  {
    printf("\nThe number is even");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

Output

```
Going to check whether a number is even or odd
Enter the number: 100
The number is even
```

Call by value and Call by reference in C

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



Let's understand call by value and call by reference in c language one by one.

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.

Let's try to understand the concept of call by value in c language by the example given below:

Call by Value Example: Swapping the values of the two variables

```
#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);
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
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

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.

Consider the following example for the call by reference.

Swapping the values of the two variables

```c
#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 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); // Formal parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference in c

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

Recursion in C

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.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```c
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    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);
    }
}
```

Output

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

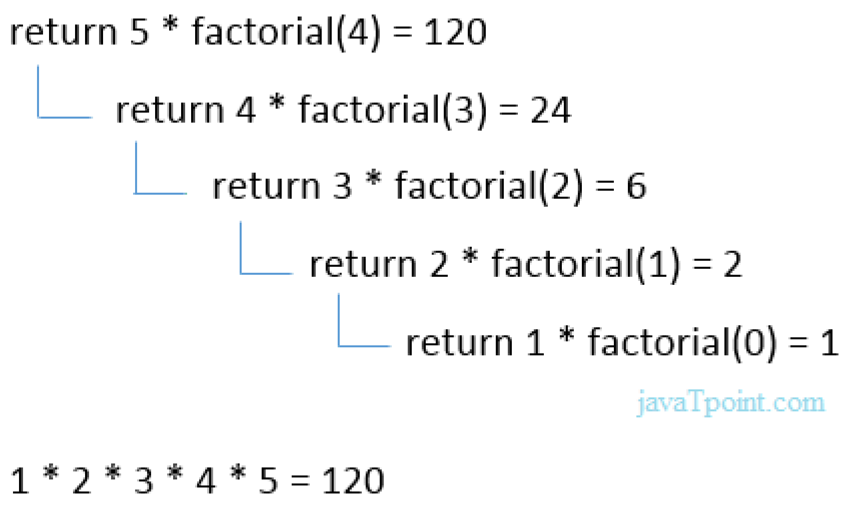

return 5 * factorial(4) = 120
return 4 * factorial(3) = 24
return 3 * factorial(2) = 6
return 2 * factorial(1) = 2
return 1 * factorial(0) = 1
javaTpoint.com
1 * 2 * 3 * 4 * 5 = 120

Fig: Recursion

Example of recursion in C

Let's see an example to find the nth term of the 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);
    }
}
```

Output

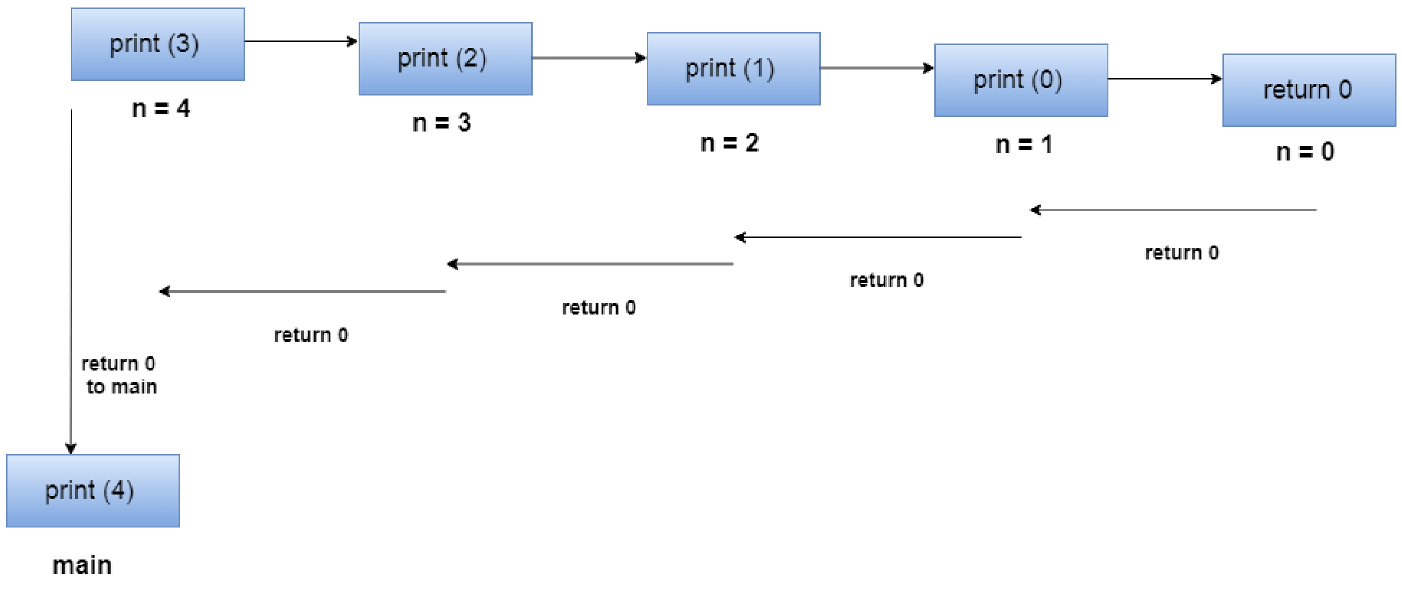
```
Enter the value of n?12
144
```

Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```
int display (int n)
{
    if(n == 0)
        return 0; // terminating condition
    else
    {
        printf("%d",n);
        return display(n-1); // recursive call
    }
}
```



Stack tracing for recursive function call

Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

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

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.

  The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

```c
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

Output:

```
garbage garbage garbage
```

Example 2

```c
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

Output:

```
 11 20 20 20 11
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```c
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

Output:

```
0 0.000000 (null)
```

Example 2

```c
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i< 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

Output:

```
10 24
11 25
12 26
```

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

```
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.

printf("%d",a);
}
```

Output:

```
0
```

Example 2

```
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
}
```

Output:

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Example 2

```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined globally, the memory will not be allocated to a
printf("%d",a);
}
```

Output

```
0
```

Example 3

```
#include <stdio.h>
int a;
int main()
{
extern int a = 0; // this will show a compiler error since we can not use extern and initializer at same time
printf("%d",a);
}
```

Output

```
compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;
```

Example 4

```
#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the pogram or not.
printf("%d",a);
}
int a = 20;
```

Output

```
20
```

Example 5

```
extern int a;
int a = 10;
#include <stdio.h>
int main()
{
printf("%d",a);
}
int a = 20; // compiler will show an error at this line
```

Output

```
compile time error
```

compile time error

C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

SN	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.

6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Command line arguments:

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"

Program name ./a.out
The argument supplied is testing1 testing2
```

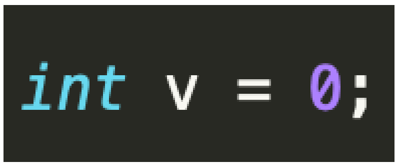
Pointer in C

What is Pointer in C?

The **Pointer** in C, is a variable that stores address of another variable. A pointer can also be used to refer to another pointer function. A pointer can be incremented/decremented, i.e., to point to the next/ previous memory location. The purpose of pointer is to save memory space and achieve faster execution time.

How to Use Pointers in C

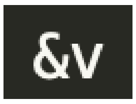
If we declare a variable v of type int, v will actually store a value.



```c
int v = 0;
```

v is equal to zero now.

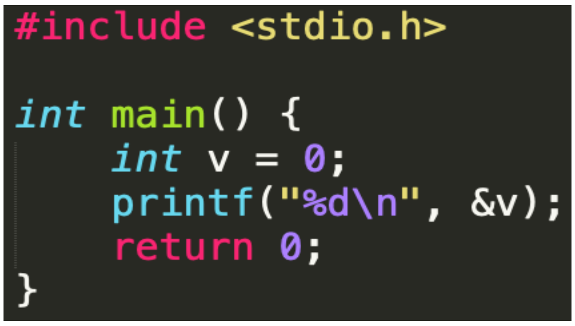
However, each variable, apart from value, also has its address (or, simply put, where it is located in the memory). The address can be retrieved by putting an ampersand (&) before the variable name.



```c
&v
```

If you print the address of a variable on the screen, it will look like a totally random number (moreover, it can be different from run to run).

Let's try this in practice with pointer in C example



```c
#include <stdio.h>

int main() {
    int v = 0;
    printf("%d\n", &v);
    return 0;
}
```

The output of this program is -480613588.

Now, what is a pointer? Instead of storing a value, a pointer will y store the address of a variable.

Pointer Variable

Int *y = &v;

VARIABLE	POINTER
A **value** stored in a **named** storage/memory address	A **variable** that **points to** the storage/mem of **another** variable

Declaring a Pointer

Like variables, pointers in C programming have to be declared before they can be used in your program. Pointers can be named anything you want as long as they obey C's naming rules. A pointer declaration has the following form.

```
data_type * pointer_variable_name;
```

Here,

- **data_type** is the pointer's base type of C's variable types and indicates the type of the variable that the pointer points to.
- The asterisk (*: the same asterisk used for multiplication) which is indirection operator, declares a pointer.

Let's see some valid pointer declarations in this C pointers tutorial:

```
int    *ptr_thing;            /* pointer to an integer */
int *ptr1,thing;/* ptr1 is a pointer to type integer and thing
is an integer variable */
double    *ptr2;    /* pointer to a double */
float    *ptr3;      /* pointer to a float */
char    *ch1 ;       /* pointer to a character */
float  *ptr, variable;/*ptr is a pointer to type float and
variable is an ordinary float variable */
```

Initialize a pointer

After declaring a pointer, we initialize it like standard variables with a variable address. If pointers in C programming are not uninitialized and used in the program, the results are unpredictable and potentially disastrous.

To get the address of a variable, we use the ampersand (&)operator, placed before the name of a variable whose address we need. Pointer initialization is done with the following syntax.

Pointer Syntax

```
pointer = &variable;
```

A simple program for pointer illustration is given below:

```
#include <stdio.h>
int main()
{
   int a=10;      //variable declaration
   int *p;        //pointer variable declaration
   p=&a;          //store address of variable a in pointer p
   printf("Address stored in a variable p is:%x\n",p);

  //accessing the address
   printf("Value stored in a variable p is:%d\n",*p);

   //accessing the value
   return 0;
}
```

Output:

```
Address stored in a variable p is:60ff08
Value stored in a variable p is:10
```

Operator	Meaning
*	Serves 2 purpose 1. Declaration of a pointer 2. Returns the value of the referenced variable
&	Serves only 1 purpose • Returns the address of a variable

Types of Pointers in C

Following are the different **Types of Pointers in C**:

1. Null Pointer

We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

Following program illustrates the use of a null pointer:

```
#include <stdio.h>
int main()
{
      int *p = NULL;   //null pointer
      printf("The value inside variable p is:\n%x",p);
      return 0;
}
```
Output:

```
The value inside variable p is:
0
```

2. Void Pointer

In C programming, a void pointer is also called as a generic pointer. It does not have any standard data type. A void pointer is created by using the keyword void. It can be used to store an address of any variable.

Following program illustrates the use of a void pointer:

```
#include <stdio.h>
int main()
{
void *p = NULL;  //void pointer
printf("The size of pointer is:%d\n",sizeof(p));
return 0;
}
```
Output:

```
The size of pointer is:4
```

3. Wild pointer

A pointer is said to be a wild pointer if it is not being initialized to anything. These types of C pointers are not efficient because they may point to some unknown memory location which may cause problems in our program and it may lead to crashing of the program. One should always be careful while working with wild pointers.

Following program illustrates the use of wild pointer:

```
#include <stdio.h>
int main()
{
int *p;     //wild pointer
printf("\n%d",*p);
return 0;
}
```

Output

```
timeout: the monitored command dumped  core
sh: line 1: 95298  Segmentation fault      timeout 10s main
```

Other types of pointers in 'c' are as follows:

- Dangling pointer
- Complex pointer
- Near pointer
- Far pointer
- Huge pointer

Direct and Indirect Access Pointers

In C, there are two equivalent ways to access and manipulate a variable content

- Direct access: we use directly the variable name
- Indirect access: we use a pointer to the variable

Let's understand this with the help of program below

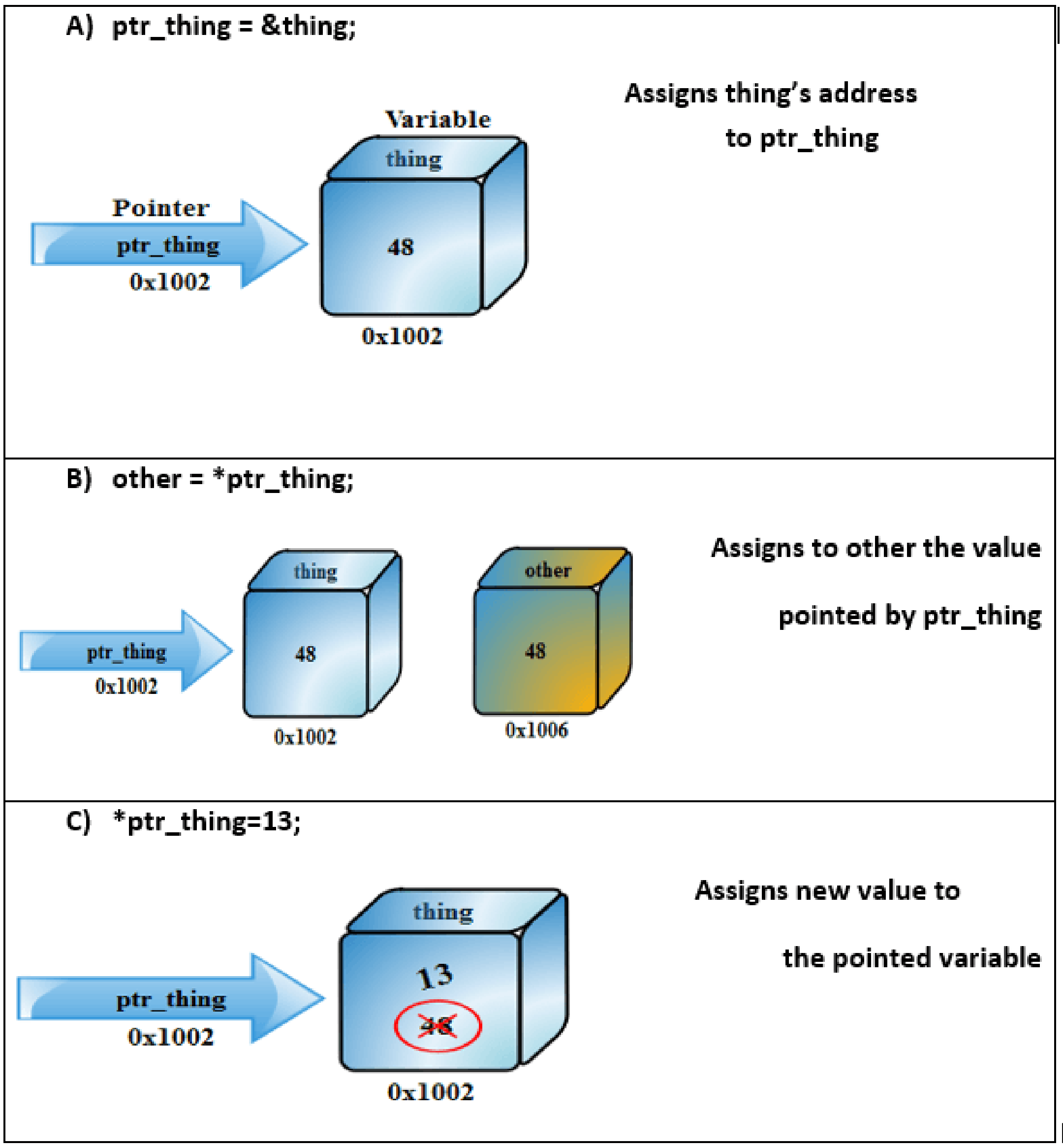
```
#include <stdio.h>
/* Declare and initialize an int variable */
int var = 1;
/* Declare a pointer to int */
int *ptr;
int main( void )
{
/* Initialize ptr to point to var */
ptr = &var;
/* Access var directly and indirectly */
printf("\nDirect access, var = %d", var);
printf("\nIndirect access, var = %d", *ptr);
/* Display the address of var two ways */
printf("\n\nThe address of var = %d", &var);
printf("\nThe address of var = %d\n", ptr);
/*change the content of var through the pointer*/
*ptr=48;
printf("\nIndirect access, var = %d", *ptr);
return 0;}
```

After compiling the program without any errors, the result is:

```
Direct access, var = 1
Indirect access, var = 1

The address of var = 4202496
The address of var = 4202496

Indirect access, var = 48
```

Pointer Arithmetics in C

The pointer operations are summarized in the following figure



Pointer Operations

Priority operation (precedence)

When working with C pointers, we must observe the following priority rules:

- The operators * and & have the same priority as the unary operators (the negation!, the incrementation++, decrement–).
- In the same expression, the unary operators *, &,!, ++, – are evaluated from right to left.

If a P pointer points to an X variable, then * P can be used wherever X can be written.

The following expressions are equivalent:

```
int X =10
int*P=&Y;
```

For the above code, below expressions are true

Expression	Equivalent Expression
Y=*P+1	Y=X+1
*P=*P+10	X=X+10
*P+=2	X+=2
++*P	++X
(*P)++	X++

In the latter case, parentheses are needed: as the unary operators * and ++ are evaluated from right to left, without the parentheses the pointer P would be incremented, not the object on which P points.

Below table shows the arithmetic and basic operation that can be used when dealing with C pointers

Operation	**Explanation**
Assignment	int *P1,*P2 P1=P2; P1 and P2 point to the same integer variable

Incrementation and decrementation	Int *P1; P1++;P1-;
Adding an offset (Constant)	This allows the pointer to move N elements in a table. The pointer will be increased or decreased by N times the number of byte(s) of the type of the variable. P1+5;

C Pointers & Arrays with Examples

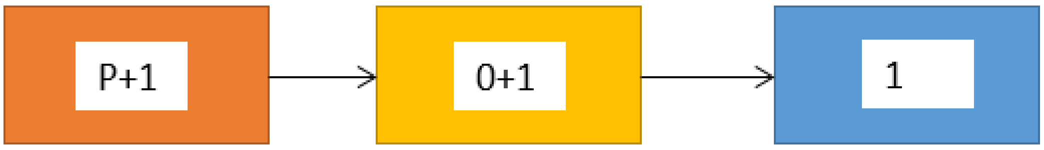
Traditionally, we access the array elements using its index, but this method can be eliminated by using pointers. Pointers make it easy to access each array element.

```
#include <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5};    //array initialization
    int *p;      //pointer declaration
                /*the ptr points to the first element of the array*/

    p=a; /*We can also type simply ptr==&a[0]  */

    printf("Printing the array elements using pointer\n");
    for(int i=0;i<5;i++)     //loop for traversing array elements
    {
            printf("\n%x",*p);   //printing array elements
            p++;    //incrementing to the next element, you can
also write p=p+1
    }
    return 0;
}
```

Output

```
1
2
3
4
5
```

Adding a particular number to a pointer will move the pointer location to the value obtained by an addition operation. Suppose p is a pointer that currently points to the memory location 0 if we perform following addition operation, p+1 then it will execute in this manner:



Pointer Addition/Increment

Since p currently points to the location 0 after adding 1, the value will become 1, and hence the pointer will point to the memory location 1.

C Pointers and Strings with Examples

A string is an array of char objects, ending with a null character '\0'. We can manipulate strings using pointers. This pointer in C example explains this section

```
#include <stdio.h>
#include <string.h>
int main()
{
char str[]="Hello Guru99!";
char *p;
p=str;
printf("First character is:%c\n",*p); p
=p+1;
printf("Next character is:%c\n",*p);
printf("Printing all the characters in a string\n");
p=str;  //reset the pointer
for(int i=0;i<strlen(str);i++)
{
printf("%c\n",*p);
p++;
}
return 0;
}
```

Output

```
First character is:H
Next character is:e
Printing all the characters in a string
H
e
l
l
o

G
u
r
u
9
9
!
```

Another way to deal strings is with an array of pointers like in the following program:

```
#include <stdio.h>
int main(){
char *materials[ ] = {  "iron",  "copper",  "gold"};
printf("Please remember these materials :\n");
int i ;
for (i = 0; i < 3; i++) {
  printf("%s\n", materials[ i ]);}
  return 0;}
```

Advantages of Pointers in C

- Pointers are useful for accessing memory locations.
- Pointers provide an efficient way for accessing the elements of an array structure.
- Pointers are used for dynamic memory allocation as well as deallocation.
- Pointers are used to form complex data structures such as linked list, graph, tree, etc.

Disadvantages of Pointers in C

- Pointers are a little complex to understand.
- Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.
- If an incorrect value is provided to a pointer, it may cause memory corruption.
- Pointers are also responsible for memory leakage.
- Pointers are comparatively slower than that of the variables.
- Programmers find it very difficult to work with the pointers; therefore it is programmer's responsibility to manipulate a pointer carefully.

Summary:

- A pointer is nothing but a memory location where data is stored.
- A pointer is used to access the memory location.
- There are various types of pointers such as a null pointer, wild pointer, void pointer and other types of pointers.
- Pointers can be used with array and string to access elements more efficiently.
- We can create function pointers to invoke a function dynamically.
- Arithmetic operations can be done on a pointer which is known as pointer arithmetic.
- Pointers can also point to function which make it easy to call different functions in the case of defining an array of pointers.
- When you want to deal different variable data type, you can use a typecast void pointer.

Structure and Union

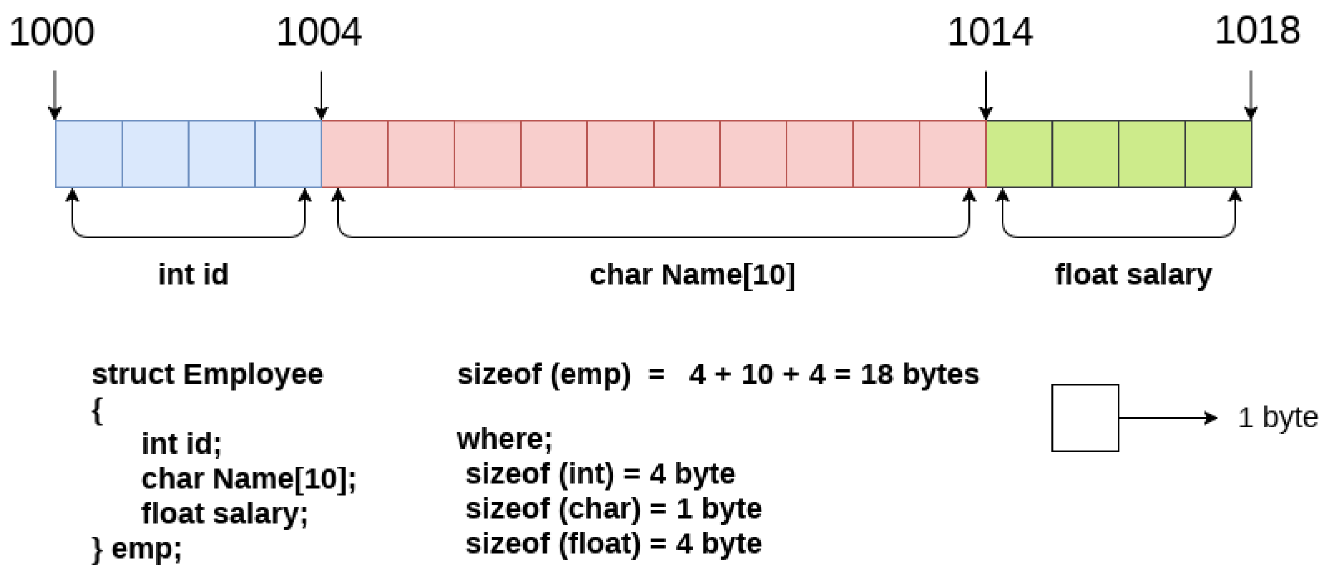
What is Structure?

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information

The **,struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeberN;
};
```

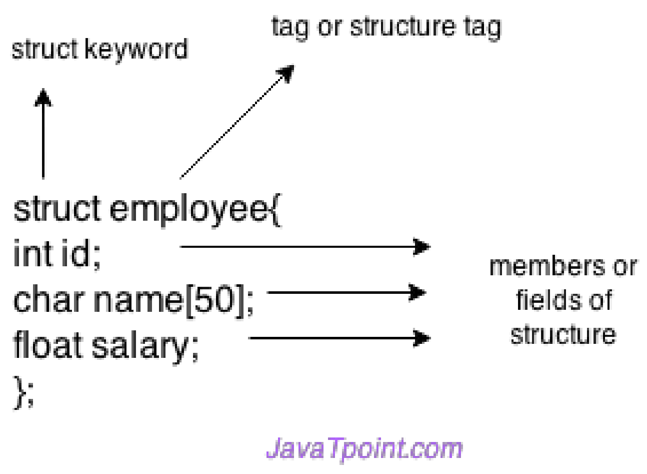
The following image shows the memory allocation of the structure employee that is defined in the above example.



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

```
sizeof (emp)  =   4 + 10 + 4 = 18 bytes

where;
 sizeof (int) = 4 byte
 sizeof (char) = 1 byte
 sizeof (float) = 4 byte
```

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



Declaring structure variable

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:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

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

Now write given code inside the main() function.

```
1. struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare 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.

Accessing members of the structure

There are two ways to access structure members:

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

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

```
p1.id
```

C Structure example

Let's see a simple example of structure in C language.

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1; //declaring e1 variable for structure
```

```c
int main( )
{
  //store first employee information
        e1.id=101;
        strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
        //printing first employee information
        printf( "employee 1 id : %d\n",  e1.id);
        printf( "employee 1 name : %s\n", e1.name);
      return 0;
      }
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
```

Let's see another example of the structure in C language to store many employees information.

```c
#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, "Sonoo Jaiswal");//copying string into char array
        e1.salary=56000;

       //store second employee information
        e2.id=102;
        strcpy(e2.name, "James Bond");
        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;
    }
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

typedef in C

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program

. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of typedef

```
typedef <existing_name> <alias_name>
```

In the above syntax, '**existing_name'** is the name of an already existing variable while '**alias name'** is another name given to the existing variable.

For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use **a typedef** keyword.

```
typedef unsigned int unit;
```

Now, we can create the variables of type **unsigned int** by writing the following statement:

```
unit a, b;
```

instead of writing: unsigned int a, b;

Let's understand through a simple example.

```
#include <stdio.h>
int main()
{
typedef unsigned int unit;
unit i,j;
i=10;
j=20;
printf("Value of i is :%d",i);
printf("\nValue of j is :%d",j);
return 0;
}
```

Output

```
Value of i is :10
Value of j is :20
```

Using typedef with structures

Consider the below structure declaration:

```
struct student

{
char name[20];
int age;
};
struct student s1;
```

In the above structure declaration, we have created the variable of **student** type by writing the following statement:

```
struct student s1;
```

The above statement shows the creation of a variable, i.e., s1, but the statement is quite big. To avoid such a big statement, we use the **typedef** keyword to create the variable of type **student**.

```
struct student
{
char name[20];
int age;
};
typedef struct student stud;
stud s1, s2;
```

In the above statement, we have declared the variable **stud** of type struct student. Now, we can use the **stud** variable in a program to create the variables of type **struct student**.

The above typedef can be written as:

```
typedef struct student
{
char name[20];
int age;
} stud;
stud s1,s2;
```

From the above declarations, we conclude that **typedef** keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

Let's see another example where we typedef the structure declaration.

```
#include <stdio.h>
typedef struct student
{
char name[20];
int age;
}stud;
int main()
{
stud s1;
printf("Enter the details of student s1: ");
printf("\nEnter the name of the student:");
scanf("%s",&s1.name);
printf("\nEnter the age of student:");
scanf("%d",&s1.age);
printf("\n Name of the student is : %s", s1.name);
printf("\n Age of the student is : %d", s1.age);
return 0;
}
```

```
 Enter the details of student s1:
 Enter the name of the student: Peter
 Enter the age of student: 28
 Name of the student is : Peter
Age of the student is : 28
```

Using typedef with pointers

We can also provide another name or alias name to the pointer variables with the help of **the typedef**.

For example, we normally declare a pointer, as shown below:

```
int* ptr;
```

We can rename the above pointer variable as given below:

```
typedef int* ptr;
```

In the above statement, we have declared the variable of type **int***. Now, we can create the variable of type **int*** by simply using the **'ptr'** variable as shown in the below statement:

```
ptr p1, p2 ;
```

In the above statement, **p1** and **p2** are the variables of type **'ptr'**.

C Array of Structures

Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
#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);
}
```

Output

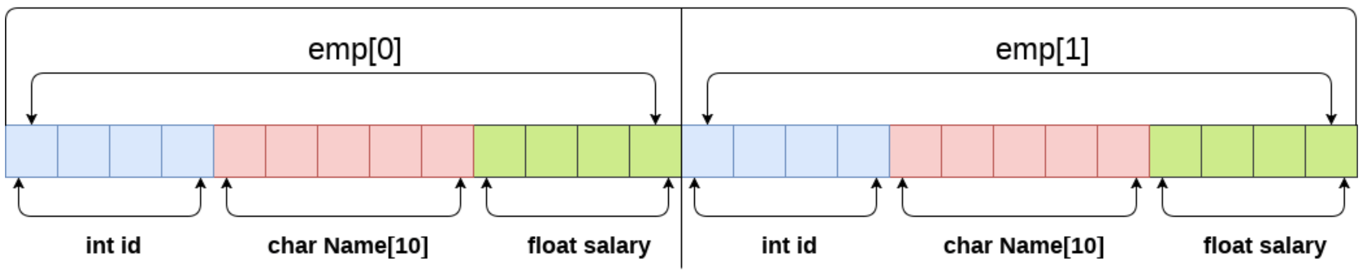
```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000
Nick 90 90.000000
```

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

Array of Structures in C

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

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
#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;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

Passing structure to 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. Consider the following example 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);
```

Union

A **union** is a special data type available in **C** that allows storing different data types in the same memory location. **union** is the keyword used to declare Union.

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

//Example

```
#include<stdio.h>
union Employee
{
        int Id;
        char Name[25];
        int Age;
        long Salary;
};
void main()
{
        union Employee E;
        printf("nEnter Employee Id : ");
        scanf("%d",&E.Id);
        printf("nEnter Employee Name : ");
        scanf("%s",&E.Name);
        printf("nEnter Employee Age : ");
        scanf("%d",&E.Age);
        printf("nEnter Employee Salary : ");
        scanf("%ld",&E.Salary);
        printf("nnEmployee Id : %d",E.Id);
        printf("nEmployee Name : %s",E.Name);
        printf("nEmployee Age : %d",E.Age);
        printf("nEmployee  Salary  : %ld",E.Salary);
}
```

//Output

```
Enter Employee Id : 102191
Enter Employee Name : Karan
Enter  Employee  Age  :  29
Enter  Employee  Salary  : 45000

Employee  Id  :  102011
Employee  Name  :  Ajay
Employee  Age  :  26
Employee Salary : 45000
```

Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

```
#include <stdio.h>
union abc
{
    int a;
    char b;
};
int main()
{
    union abc *ptr; // pointer variable declaration
    union abc var;
    var.a= 90;
    ptr = &var;
    printf("The value of a is : %d", ptr->a);
    return 0;
}
```

In the above code, we have created a pointer variable, i.e., *ptr, that stores the address of var variable. Now, ptr can access the variable 'a' by using the (->) operator. Hence the output of the above code would be 90.

Difference between Structure and Union

Parameter	Structure	Union
Keyword	**struct**	**union**
Memory	Each member has a unique Memory	All members share allocated Memory
Initialization	All members are initialized together	Only first member can be initialized
Data Access	All members is accessed together	One member is accessed at a time

Value Change	Changing the value of one member will not affect others	Changing the value of one member will affect others

File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. 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

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

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

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

```
1.  FILE *fopen( const char * filename, const char * mode );
```

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.

We can use one of the following modes in the fopen() function.

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.

Consider the following example which opens a file in write mode.

```
#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 ) ;
}
```

Output

The content of the file will be printed.

```
#include;
void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and
stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

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

Writing File : fprintf() function

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

Syntax:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

Example:

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

Reading File : fscanf() function

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>
main(){
   FILE *fp;
   char buff[255];//creating char array to store data of file
   fp = fopen("file.txt", "r");
   while(fscanf(fp, "%s", buff)!=EOF){
   printf("%s ", buff );
   }
   fclose(fp);
        }
```

Output:

```
Hello file by fprintf...
```

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.

Syntax:

```
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("This is javatpoint", fp);

   fseek( fp, 7, SEEK_SET );
   fputs("sonoo jaiswal", fp);
          fclose(fp);
       }
```

myfile.txt

```
This is sonoo Jaiswal
```

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)
```

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:

1. **long int** ftell(**FILE** *stream)