

C TOKENS:

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual units in a C program are known as C tokens.

C tokens are of six types. They are,

1. Keywords (eg: int, while),
2. Identifiers (eg: total,number1),
3. Constants (eg: 10, 20),
4. Strings (eg: "total", "hello"),
5. Special symbols (eg: (), {}),
6. Operators (eg: +, /,-,*)

C TOKENS EXAMPLE PROGRAM:

```
int main()
{
    int x, y, total;
    x = 10, y = 20;
    total = x + y;
    printf ("Total = %d \n", total);
}
```

where,

- {, }, (,) – delimiter
- int – keyword
- x, y, total – identifier
- {, }, (,), int, x, y, total – tokens

1. IDENTIFIERS IN C LANGUAGE:

- Each program elements in a C program are given a name called identifiers.
- Names given to identify Variables, functions and arrays are examples for identifiers. eg. x is a name given to integer variable in above program.

RULES FOR CONSTRUCTING IDENTIFIER NAME IN C:

1. First character should be an alphabet or underscore.
2. Succeeding characters might be digits or letter.
3. Punctuation and special characters aren't allowed except underscore.
4. Identifiers should not be keywords.

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

void

Invalid as it is a keyword

2. KEYWORDS IN C LANGUAGE:

- Keywords are pre-defined words in a C compiler.
- Each keyword is meant to perform a specific function in a C program.
- Since keywords are referred names for compiler, they can't be used as variable name.

C language supports 32 keywords which are given below. Click on each keywords below for detail description and example programs.

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

3.C Constants

C constants refers to the data items that do not change their value during the program execution. Several types of C constants that are allowed in C are:

1. Integer Constants

Integer constants are whole numbers without any fractional part. It must have at least one digit and may contain either + or – sign. A number with no sign is assumed to be positive.

There are three types of integer constants:

1.1. Decimal Integer Constants

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

Example of valid decimal integer constants

341, -341, 0, 8972

1.2. Octal Integer Constants

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

Example of valid octal integer constants

010, 0424, 0, 0540

1.3. Hexadecimal Integer Constants

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

Example of valid hexadecimal integer constants

0xD, 0X8d, 0X, 0xbD

It should be noted that, octal and hexadecimal integer constants are rarely used in programming.

2. Real Constants

The numbers having fractional parts are called real or floating point constants. These may be represented in one of the two forms called *fractional form* or the *exponent form* and may also have either + or – sign preceding it.

Example of valid real constants in fractional form or decimal notation

0.05, -0.905, 562.05, 0.015

Representing a real constant in exponent form

The general format in which a real number may be represented in exponential or scientific form is

mantissa e exponent

The mantissa must be either an integer or a real number expressed in decimal notation.

The letter e separating the mantissa and the exponent can also be written in uppercase i.e. E

And, the exponent must be an integer.

Examples of valid real constants in exponent form are:

252E85, 0.15E-10, -3e+8

3. Character Constants

A character constant contains one single character enclosed within single quotes.

Examples of valid character constants

'a' , 'Z' , '5'

It should be noted that character constants have numerical values known as ASCII values, for example, the value of 'A' is 65 which is its ASCII value.

4.String Constants

String constants are sequence of characters enclosed within double quotes. For example,

"hello"

"abc"

"hello911"

Every sting constant is automatically terminated with a special character '' called the null character which represents the end of the string.

For example, "hello" will represent "hello" in the memory.

Thus, the size of the string is the total number of characters plus one for the null character.

HOW TO USE CONSTANTS IN A C PROGRAM?

We can define constants in a C program in the following ways.

1. By "const" keyword
2. By "#define" preprocessor directive

EXAMPLE PROGRAM USING CONST KEYWORD IN C:

```
#include<stdio.h>
void main()
{
constint height = 100; /*int constant*/
const float number = 3.14; /*Real constant*/
const char letter = 'A'; /*char constant*/
const char letter_sequence[10] = "ABC"; /*string constant*/
const char backslash_char = '\\'; /*special char cnst*/
printf("value of height :%d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n", letter_sequence);

printf("value of backslash_char : %c \n", backslash_char);
}
o/p:-
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

4.Special Symbols

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

[] () {} , ; : * ... = #

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

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

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

5.Data types

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

The types in C can be classified as follows –

S.N.	Types & Description
1	Basic Types They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types (c)char (d) double
2	Enumerated types They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	The type void The type specifier <i>void</i> indicates that no value is available.
4	Derived types They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges –

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsignedint	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	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

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes. Given below is an example to get the size of int type on any machine –

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

int main(){

printf("Storage size for int : %d \n",sizeof(int));

return0;
```

```
}
```

When you compile and execute the above program, it produces the following result on Linux –

Storage size for int : 4

Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values –

The void Type

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

C Qualifiers

Qualifiers alters the meaning of base data types to yield a new data type.

Size qualifiers

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

```
long double i;
```

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

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

Sign qualifiers

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

```
// unsigned variables cannot hold negative value
```

```
unsigned int positiveInteger;
```

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

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

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

Constant qualifiers

An identifier can be declared as a constant. To do so `const` keyword is used.

```
const int cost = 20;
```

The value of `cost` cannot be changed in the program.

Volatile qualifiers

A variable should be declared volatile whenever its value can be changed by some external sources outside the program. Keyword `volatile` is used for creating volatile variables

6.C - Operators

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

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

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

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by 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 –

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 –

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 –

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 Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e., 1100 0011 in 2's complement form.
<<	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 –

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

sizeof & ternary

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

Operator	Description	Example
----------	-------------	---------

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

Variable declaration and definition

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic data types, there will be the following basic variable types –

Type	Description
char	Typically a single octet(one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use the keyword **extern** to declare a variable

at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

```
type variable_list;
```

Some valid declarations are shown here –

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

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

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows –

```
type variable_name = value;
```

Here, **type** must be a valid C data type including `char`, `int`, `float`, `double` or any user-defined object; and **variable_name** identifier name.

Some examples are –

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

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

Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function –

```
#include<stdio.h>  
  
int main ()  
{  
    // Variable declaration:  
    int a, b;  
    int c;  
    float f;  
    /* actual initialization */  
    a=10;  
    b=20;
```

```
c= a + b;
printf("value of c : %d \n", c);

f=70.0/3.0;
printf("value of f : %f \n", f);

return 0;
}
```

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

```
value of c : 30
value of f : 23.333334
```

C Programming Input Output (I/O): printf() and scanf()

This section focuses on two in-built functions printf() and scanf() to perform I/O task in C programming.

Programming Input Output

C programming has several in-built library functions to perform input and output tasks.

Two commonly used functions for I/O (Input/Output) are printf() and scanf().

The scanf() function reads formatted input from standard input (keyboard) whereas the printf() function sends formatted output to the standard output (screen).

Example #1: C Output

```
#include<stdio.h>    //This is needed to run printf() function.

int main()
{
printf("C Programming"); //displays the content inside quotation
return 0;
}
```

Output: C Programming

How this program works?

All valid C program must contain the main() function. The code execution begins from the start of main() function.

The printf() is a library function to send formatted output to the screen. The printf() function is declared in "stdio.h" header file.

Here, `stdio.h` is a header file (standard input output header file) and `#include` is a preprocessor directive to paste the code from the header file when necessary. When the compiler encounters `printf()` function and doesn't find `stdio.h` header file, compiler shows error.

The `return 0;` statement is the "Exit status" of the program. In simple terms, program ends.

Example #2: C Integer Output

```
#include<stdio.h>

int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```

Output:Number = 5

Inside the quotation of `printf()` function, there is a format string `"%d"` (for integer). If the format string matches the argument (`testInteger` in this case), it is displayed on the screen.

Example #3: C Integer Input/Output

```
#include<stdio.h>

int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d",&testInteger);
    printf("Number = %d",testInteger);
    return 0;
}
```

Output

Enter an integer: 4

Number = 4

The `scanf()` function reads formatted input from the keyboard. When user enters an integer, it is stored in variable `testInteger`.

Note the `'&'` sign before `testInteger`; `&testInteger` gets the address of `testInteger` and the value is stored in that address.

Example #3: C Floats Input/Output

```
#include<stdio.h>

int main()
{
float f;

printf("Enter a number: ");

// %f format string is used in case of floats

scanf("%f",&f);

printf("Value = %f", f);

return 0;

}
```

Output

Enter a number: 23.45

Value = 23.450000

The format string "%f" is used to read and display formatted in case of floats.

Example #4: C Character I/O

```
#include<stdio.h>

int main()
{
charchr;

printf("Enter a character: ");

scanf("%c",&chr);

printf("You entered %c.",chr);

return 0;

}
```

Output

Enter a character: g

You entered g.

Format string %c is used in case of character types.

Little bit on ASCII code

A character in C programming language is stored as a particular integer in memory location. The integer value corresponding to a character is known as its ASCII value (American Standard Code for Information Interchange). For example, the ASCII value of 'A' is 65. A character and its ASCII value can be used interchangeably. That's why we can perform all arithmetic operations on characters like 'A' + 3, 'A' / 4 etc. If any expression contains a character then its corresponding ASCII value is used in expression. When we store a character in a variable of data type char, the ASCII value of character is stored instead of that character itself.

Example #5: C ASCII Code

```
#include<stdio.h>

int main()
{
    char chr;

    printf("Enter a character: ");

    scanf("%c",&chr);

    // When %c text format is used, character is displayed in case of character types

    printf("You entered %c.\n",chr);

    // When %d text format is used, integer is displayed in case of character types

    printf("ASCII value of %c is %d.", chr, chr);

    return 0;
}
```

Output

Enter a character: g

You entered g.

ASCII value of g is 103.

The ASCII value of character 'g' is 103. When, 'g' is entered, 103 is stored in variable var1 instead of g.

You can display a character if you know ASCII code of that character. This is shown by following example.

Example #6: C ASCII Code

```
#include<stdio.h>

int main()
{
    int chr = 69;

    printf("Character having ASCII value 69 is %c.",chr);
}
```

```
return 0;
```

```
}
```

Output

Character having ASCII value 69 is E.

More on Input/Output of floats and Integers

Integer and floats can be displayed in different formats in C programming.

Example #7: I/O of Floats and Integers

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

```
int main()
```

```
{
```

```
int integer = 9876;
```

```
float decimal = 987.6543;
```

```
    // Prints the number right justified within 6 columns
```

```
printf("4 digit integer right justified to 6 column: %6d\n", integer);
```

```
    // Tries to print number right justified to 3 digits but the number is not right adjusted because there are only 4 numbers
```

```
printf("4 digit integer right justified to 3 column: %3d\n", integer);
```

```
    // Rounds to two digit places
```

```
printf("Floating point number rounded to 2 digits: %.2f\n",decimal);
```

```
    // Rounds to 0 digit places
```

```
printf("Floating point number rounded to 0 digits: %.f\n",987.6543);
```

```
    // Prints the number in exponential notation(scientific notation)
```

```
printf("Floating point number in exponential form: %e\n",987.6543);
```

```
return 0;
```

```
}
```

Output

4 digit integer right justified to 6 column: 9876

4 digit integer right justified to 3 column: 9876

Floating point number rounded to 2 digits: 987.65

Floating point number rounded to 0 digits: 988

Floating point number in exponential form: 9.876543e+02

ASCII table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Precedence and associativity

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of type) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right

	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right
<p>Note 1: Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.</p> <p>Note 2: Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement <code>y = x * z++</code>; the current value of <code>z</code> is used to evaluate the expression (<i>i.e.</i>, <code>z++</code> evaluates to <code>z</code>) and <code>z</code> only incremented after all else is done.</p>		

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.

Example

Try the following example to understand operator precedence in C –

```
#include<stdio.h>

main()
{
int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;

e = (a + b) * c / d;    // ( 30 * 15 ) / 5
printf("Value of (a + b) * c / d is : %d\n", e);

e = ((a + b) * c) / d;  // (30 * 15 ) / 5
printf("Value of ((a + b) * c) / d is : %d\n", e);

e = (a + b) * (c / d);  // (30) * (15/5)
printf("Value of (a + b) * (c / d) is : %d\n", e);

e = a + (b * c) / d;    // 20 + (150/5)
```

```
printf("Value of a + (b * c) / d is : %d\n" , e );
return 0;
}
```

Output:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
```

Summary of operator precedence

1. **Comma Operator Has Lowest Precedence.**
2. **Unary Operators** are Operators having **Highest Precedence.**
3. **Sizeof** is Operator not Function.
4. Operators sharing Common Block in the Above Table have Equal Priority or Precedence.
5. While Solving Expression, Equal Priority Operators are handled on the basis of FIFO [First in First Out] i.e Operator Coming First is handled First.

Important definitions

Unary Operator	<p>A unary operation is an operation with only one operand, i.e. an operation with a single input .</p> <p>A unary operator is one which has only one operand. e.g. post / pre increment operator</p>
Binary Operator	A Binary operator is one which has two operand. e.g. plus , Minus .
Associativity	Their associativity indicates in what order operators of equal precedence in an expression are applied
Precedence	Priority Of Operator

Examples : operator precedence & associativity

We have listed out some of the examples based on operator precedence & associativity. Examples will give you clear picture how to use operator precedence & associativity table chart while solving the expression

Example 1 : Simple use of precedence chart

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

```

{
int num1 =10, num2 =20;
int result;
result= num1 *2+ num2;
printf("\nResult is : %d", result);
return(0);
}

```

consider the simple expression used in above program and refer operator precedence & associativity table chart

```
result = num1 * 2 + num2;
```

In this case multiplication operator will have higher priority than addition and assignment operator so multiplication will be evaluated firstly.

```

result = num1 * 2 + num2;
result = 10 * 2 + 20;
result = 20 + 20;
result = 40;

```

Example 2 : Use of associativity

In above example none of the operator has equal priority. In the below example some operators are having same priority.(left to right)

```
result = num1 * 2 + num2 * 2 ;
```

In the above expression we have overall 3 operators i.e. 2 multiplication, 1 addition and 1 assignment operator.

```

result = num1 * 2 + num2 * 2 ;
result = 10 * 2 + 20 * 2 ;
result = 20 + 20 * 2 ;
result = 20 + 40 ;
result = 60 ;

```


if statement

```
if (testExpression)
{
    // statements
}
```

The if statement evaluates the test expression inside the parenthesis.

If the test expression is evaluated to true (nonzero), statements inside the body of if is executed.

If the test expression is evaluated to false (0), statements inside the body of if is skipped from execution.

To learn more on when test expression is evaluated to nonzero (true) and 0 (false), check out [relational operators](#) and [logical operators](#).

Flowchart of if statement

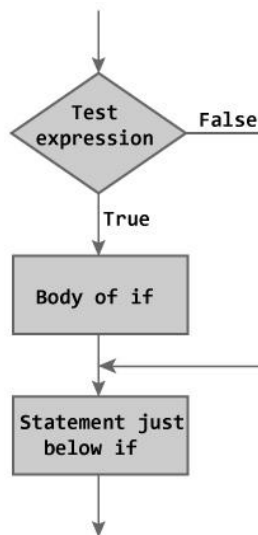


Figure: Flowchart of if Statement

Example #1: C if statement

```
// Program to display a number if user enters negative number
// If user enters positive number, that number won't be displayed
```

```
#include <stdio.h>
int main()
{
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    // Test expression is true if number is less than 0
    if (number < 0)
    {
        printf("You entered %d.\n", number);
    }
}
```

```
printf("The if statement is easy.");  
return 0;  
}
```

Output 1

Enter an integer: -2

You entered -2.

The if statement is easy.

When user enters -2, the test expression (number < 0) becomes true. Hence, You entered -2 is displayed on the screen.

Output 2

Enter an integer: 5

The if statement in C programming is easy.

When user enters 5, the test expression (number < 0) becomes false and the statement inside the body of if is skipped.

C if...else statement

The if...else statement executes some code if the test expression is true (nonzero) and some other code if the test expression is false (0).

Syntax of if...else

```
if (testExpression) {  
    // codes inside the body of if  
}  
else {  
    // codes inside the body of else  
}
```

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

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

Flowchart of if...else statement

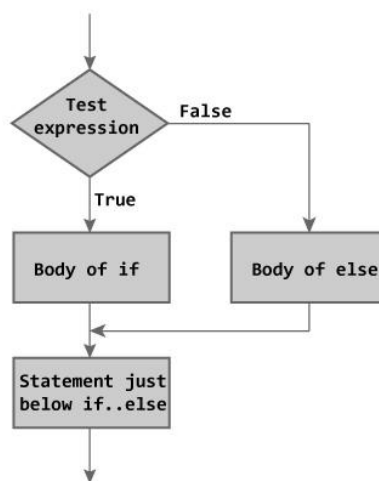


Figure: Flowchart of if...else Statement

Example #2: C if...else statement

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

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

```
int main()
```

```
{
```

```
    int number;
```

```
    printf("Enter an integer: ");
```

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

```
    // True if remainder is 0
```

```
    if( number%2 == 0 )
```

```
        printf("%d is an even integer.",number);
```

```
    else
```

```
        printf("%d is an odd integer.",number);
```

```
    return 0;
```

```
}
```

Output

Enter an integer: 7

7 is an odd integer.

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

Nested if...else statement (if...elseif....else Statement)

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The nested if...else statement allows you to check for multiple test expressions and execute different codes for more than two conditions.

Syntax of nested if...else statement.

```
if (testExpression1)
```

```
{
```

```
    // statements to be executed if testExpression1 is true
```

```
}
```

```
else if(testExpression2)
```

```
{
```

```
    // statements to be executed if testExpression1 is false and testExpression2 is true
```

```
}
```

```
else if (testExpression 3)
```

```
{
```

```
    // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true
```

```
}
```

```
.
```

```
.
```

```
else
```

```
{
```

```
    // statements to be executed if all test expressions are false
```

```
}
```

Example #3: C Nested if...else statement

```
// Program to relate two integers using =, > or <
#include <stdio.h>
int main()
{
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if two integers are equal.
    if(number1 == number2)
    {
        printf("Result: %d = %d",number1,number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2)
    {
        printf("Result: %d > %d", number1, number2);
    }

    // if both test expression is false
    else
    {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

Output

Enter two integers: 12

23

Result: 12 < 23

You can also use switch statement to make decision between multiple possibilites.

Try out these examples to learn more:

- Check Whether a Number is Even or Odd
- Check Whether a Character is Vowel or Consonant
- Find the Largest Number Among Three Numbers

LOOP CONTROL STRUCTURES

Loops are used in programming to repeat a specific block until some end condition is met. There are three loops in C programming:

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

for Loop

The syntax of for loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
    // codes
}
```

How for loop works?

The initialization statement is executed only once.

Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of for loop is executed and the update expression is updated.

This process repeats until the test expression is false.

The for loop is commonly used when the number of iterations is known.

To learn more on test expression (when test expression is evaluated to nonzero (true) and 0 (false)), one should now the relational and logical operators.

for loop Flowchart

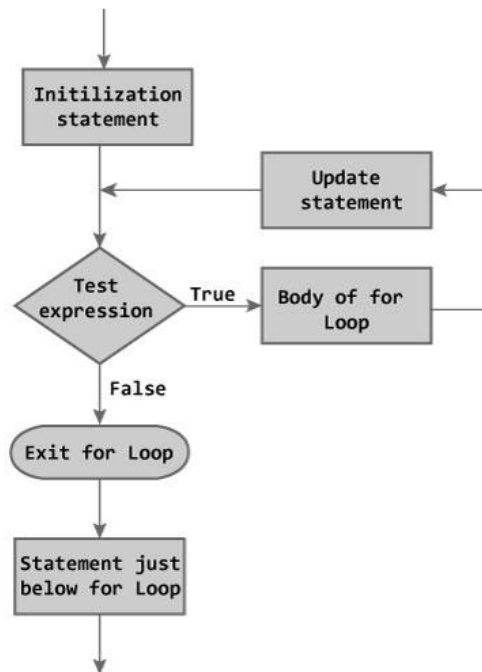


Figure: Flowchart of for Loop

Example: for loop

// Program to calculate the sum of first n natural numbers

// Positive integers 1,2,3...n are known as natural numbers

```
#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when n is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

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

    return 0;
}
```

Output

Enter a positive integer: 10

Sum = 55

The value entered by the user is stored in variable num. Suppose, the user entered 10.

The count is initialized to 1 and the test expression is evaluated. Since, the test expression count <= num (1 less than or equal to 10) is true, the body of for loop is executed and the value of sum will equal to 1.

Then, the update statement ++count is executed and count will equal to 2. Again, the test expression is evaluated. Since, 2 is also less than 10, the test expression is evaluated to true and the body of for loop is executed. Now, the sum will equal 3.

This process goes on and the sum is calculated until the count reaches 11.

When the count is 11, the test expression is evaluated to 0 (false) as 11 is not less than or equal to 10. Therefore, the loop terminates and next, the total sum is printed.

Try out these examples to learn more:

- Calculate the Sum of Natural Numbers
- Find Factorial of a Number
- Generate Multiplication Table

while loop

The syntax of a while loop is:

```
while (testExpression)
{
    //codes
}
```

where, testExpression checks the condition is true or false before each loop.

How while loop works?

The while loop evaluates the test expression.

If the test expression is true (nonzero), codes inside the body of while loop are executed. The test expression is evaluated again. The process goes on until the test expression is false.

When the test expression is false, the while loop is terminated.

Flowchart of while loop

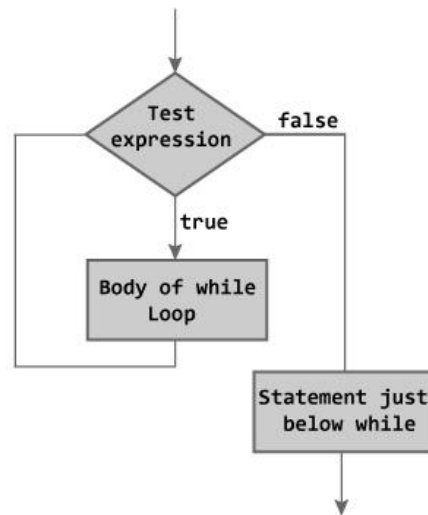


Figure: Flowchart of while Loop

Example #1: while loop

// Program to find factorial of a number

// For a positive integer n, factorial = 1*2*3...n

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

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

    factorial = 1;

    // loop terminates when number is less than or equal to 0
    while (number > 0)
    {
        factorial *= number; // factorial = factorial*number;
        --number;
    }

    printf("Factorial= %lld", factorial);

    return 0;
}
```

Output

Enter an integer: 5

Factorial = 120

To learn more on test expression (when test expression is evaluated to nonzero (true) and 0 (false)), one must know relational and logical operators.

do...while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed once, before checking the test expression. Hence, the do...while loop is executed at least once.

do...while loop Syntax

```
do
{
    // codes
}
while (testExpression);
```

How do...while loop works?

The code block (loop body) inside the braces is executed once.

Then, the test expression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to 0 (false).

When the test expression is false (nonzero), the do...while loop is terminated.

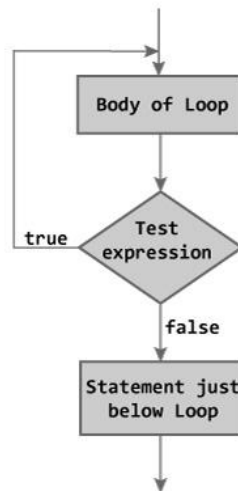


Figure: Flowchart of do...while Loop

Example #2: do...while loop

// Program to add numbers until user enters zero

```
#include <stdio.h>
int main()
{
    double number, sum = 0;

    // loop body is executed at least once
```



```

do
{
    printf("Enter a number: ");
    scanf("%lf", &number);
    sum += number;
}
while(number != 0.0);

printf("Sum = %.2lf",sum);

return 0;
}

```

Output

Enter a number: 1.5

Enter a number: 2.4

Enter a number: -3.4

Enter a number: 4.2

Enter a number: 0

Sum = 4.70

Try out these examples to learn more:

- Check Whether a Character is Vowel or Consonant
- Calculate the Sum of Natural Numbers
- Display Fibonacci Sequence

C Programming break and continue Statement

It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression. In such cases, break and continue statements are used.

break Statement

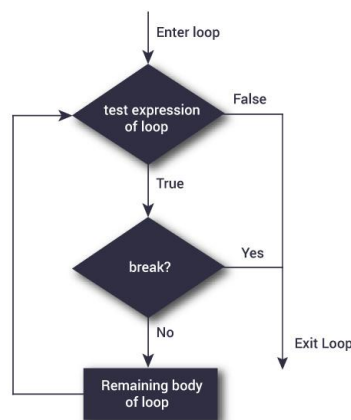
The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used with decision making statement such as if...else.

Syntax of break statement

```
break;
```

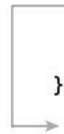
The simple code above is the syntax for break statement.

Flowchart of break statement




How break statement works?

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



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



Example #1: break statement

// Program to calculate the sum of maximum of 10 numbers

// Calculates sum until user enters positive number

```
# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;

    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        // If user enters negative number, loop is terminated
        if(number < 0.0)
        {
            break;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

Output

Enter a n1: 2.4

Enter a n2: 4.5

Enter a n3: 3.4

Enter a n4: -3

Sum = 10.30

This program calculates the sum of maximum of 10 numbers. It's because, when the user enters negative number, the break statement is executed and loop is terminated.

In C programming, break statement is also used with switch...case statement.

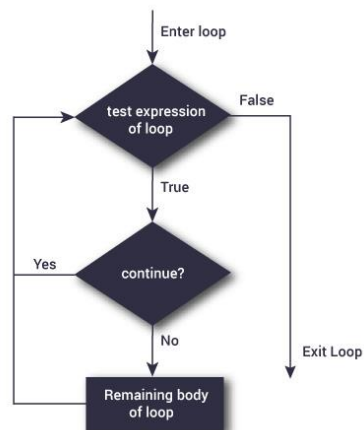
continue Statement

The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else.

Syntax of continue Statement

continue;

Flowchart of continue Statement



How continue statement works?

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

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

Example #2: continue statement

// Program to calculate sum of maximum of 10 numbers
// Negative numbers are skipped from calculation

```
# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;

    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        // If user enters negative number, loop is terminated
        if(number < 0.0)
        {
            continue;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

Output

```
Enter a n1: 1.1
Enter a n2: 2.2
Enter a n3: 5.5
Enter a n4: 4.4
Enter a n5: -3.4
Enter a n6: -45.5
Enter a n7: 34.5
Enter a n8: -4.2
Enter a n9: -1000
Enter a n10: 12
Sum = 59.70
```

In the program, when the user enters positive number, the sum is calculated using `sum += number;` statement. When the user enters negative number, the `continue` statement is executed and skips the negative number from calculation.

Try out these examples to learn more:

- Display Fibonacci Sequence

- Check Whether a Number is Prime or Not
- Display Prime Numbers Between Two Intervals

C switch...case Statement

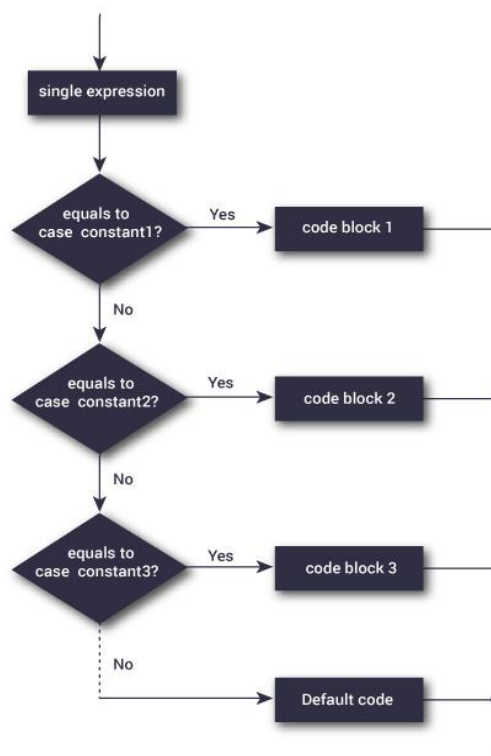
The if..else..if ladder allows you to execute a block code among many alternatives. If you are checking on the value of a single variable in if...else...if, it is better to use switch statement. The switch statement is often faster than nested if...else (not always). Also, the syntax of switch statement is cleaner and easy to understand.

Syntax of switch...case

```
switch (n)
{
    case constant1:
        // code to be executed if n is equal to constant1;
        break;
    case constant2:
        // code to be executed if n is equal to constant2;
        break;
    .
    .
    .
    default:
        // code to be executed if n doesn't match any constant
}
```

When a case constant is found that matches the switch expression, control of the program passes to the block of code associated with that case. In the above pseudocode, suppose the value of n is equal to constant2. The compiler will execute the block of code associate with the case statement until the end of switch block, or until the break statement is encountered. The break statement is used to prevent the code running into the next case.

switch Statement Flowchart



Example: switch Statement

```
// Program to create a simple calculator
// Performs addition, subtraction, multiplication or division depending the input from user
#include <stdio.h>
int main() {

    char operator;
    double firstNumber,secondNumber;

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);

    printf("Enter two operands: ");
    scanf("%lf %lf",&firstNumber, &secondNumber);

    switch(operator)
    {
        case '+':
            printf("%.1lf + %.1lf = %.1lf",firstNumber, secondNumber, firstNumber+secondNumber);
            break;

        case '-':
            printf("%.1lf - %.1lf = %.1lf",firstNumber, secondNumber, firstNumber-secondNumber);
            break;

        case '*':
            printf("%.1lf * %.1lf = %.1lf",firstNumber, secondNumber, firstNumber*secondNumber);
            break;

        case '/':
            printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber, firstNumber/firstNumber);
            break;

        // operator is doesn't match any case constant (+, -, *, /)
        default:
            printf("Error! operator is not correct");
    }

    return 0;
}
```

Output

```
Enter an operator (+, -, *, /): -
Enter two operands: 32.5
12.4
32.5 - 12.4 = 20.1
```

The - operator entered by the user is stored in operator variable. And, two operands 32.5 and 12.4 are stored in variables firstNumber and secondNumber respectively.

Then, control of the program jumps to

```
printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber, firstNumber/firstNumber);
```

Finally, the break statement ends the switch statement.

If break statement is not used, all cases after the correct case is executed.

Try out these examples to learn more:

- Make a Simple Calculator Using switch...case