What Is Comment In C Language?

A comment is an explanation or description of the source code of the program. It helps a developer explain logic of the code and improves program readability. At run-time, a comment is ignored by the compiler.

There are two types of comments in C:

1) A comment that starts with a slash asterisk /* and finishes with an asterisk slash */ and you can place it anywhere in your code, on the same line or several lines.

2) Single-line Comments which uses a double slash // dedicated to comment single lines

Example Single Line Comment

```
// single line comment example
```

Here is an example of comments type

```
// C program to demo

// Single Line comment

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

        // This is a single line comment
        printf("ICS");
        return 0;  // return zero
}
```

Example Multi Line Comment

```
/* Sample Multiline Comment

Line 1

Line 2

….

…

*/
```

Example Multi Line Comment

```
#include <stdio.h>
int main() {
/* in main function
I can write my principal code
And this in several comments line */
int x = 42; //x is a integer variable
printf("%d", x);
return 0;
}
```

Why do you need comments?

So, it is highly recommended to insert comments to your code because it is good programming practice. Comments do not affect a program because the compiler ignores them.

Comments help the developer understand the logic/algorithm of the code if he revisits it after a long time.

C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

int money;

Here, int is a keyword that indicates money is a variable of type int (integer).

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

C Keywords

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

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.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

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, _m _, sum_1, etc.

Example of invalid identifiers

- 2sum (starts with a numerical digit)
- int (reserved word)
- char (reserved word)
- m+n (special character, i.e., '+')

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.

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;
}
```

Output

```
Value of a is : 10

Value of A is :20
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

Data Types in C

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

C Data Types

There are the following data types in C language.

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

Basic Data Types

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:

Following are the examples of some very common data types used in C:

char: The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

int: As the name suggests, an int variable is used to store an integer.

float: It is used to store decimal numbers (numbers with floating point value) with single precision.

double: It is used to store decimal numbers (numbers with floating point value) with double precision.

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

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

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

Type	Format Specifier
int	%d, %i
char	%c
float	%f
double	%lf

short int	%hd
unsigned int	%u
long int	%ld, %li
long long int	%lld, %lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10

We can use int for declaring an integer variable.

int id;

Here, id is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

int id, age;

The size of int is usually 4 bytes (32 bits). And, it can take 232 distinct states from -2147483648 to 2147483647.

float and double

float and double are used to hold real numbers.

float salary;

double price;

In C, floating-point numbers can also be represented in exponential. For example, *float normalizationFactor = 22.442e2;*

What's the difference between float and double?

- double has 2x more precision then float.
- float is a 32 bit IEEE 754 single precision Floating Point Number1 bit for the sign, (8 bits for the exponent, and 23* for the value), i.e. float has 7 decimal digits of precision.
- double is a 64 bit IEEE 754 double precision Floating Point Number (1 bit for the sign, 11 bits for the exponent, and 52* bits for the value), i.e. double has 15 decimal digits of precision.

```
// C program to demonstrate double and float precision values

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

// utility function which calculate roots of  quadratic equation using double values
void double_solve(double a, double b, double c){
	double d = b*b - 4.0*a*c;
	double sd = sqrt(d);
	double r1 = (-b + sd) / (2.0*a);
	double r2 = (-b - sd) / (2.0*a);
	printf("%.5f\t%.5f\n", r1, r2);
}

// utility function which calculate roots of  quadratic equation using float values
void float_solve(float a, float b, float c){
	float d = b*b - 4.0f*a*c;
	float sd = sqrtf(d);
	float r1 = (-b + sd) / (2.0f*a);
	float r2 = (-b - sd) / (2.0f*a);
	printf("%.5f\t%.5f\n", r1, r2);
}

// driver program
int main(){
	float fa = 1.0f;
	float fb = -4.0000000f;
	float fc = 3.9999999f;
	double da = 1.0;
	double db = -4.0000000;
```

```
        double dc = 3.9999999;

        printf("roots of equation x2 - 4.0000000 x + 3.9999999 = 0 are : \n");
        printf("for float values: \n");
        float_solve(fa, fb, fc);

        printf("for double values: \n");
        double_solve(da, db, dc);
        return 0;
}
```

char

Keyword char is used for declaring character type variables. For example,

char test = 'h';

The size of the character variable is 1 byte.

void

void is an incomplete type. It means "nothing" or "no type". You can think of void as absent.

For example, if a function is not returning anything, its return type should be void.

Note that, you cannot create variables of void type.

short and long

If you need to use a large number, you can use a type specifier long. Here's how:

long a;

long long b;

long double c;

Here variables a and b can store integer values. And, c can store a floating-point number.

If you are sure, only a small integer ([−32,767, +32,767] range) will be used, you can use short.

short d;

You can always check the size of a variable using the sizeof() operator.

```
#include <stdio.h>
int main() {
  short a;
  long b;
  long long c;
  long double d;

  printf("size of short = %d bytes\n", sizeof(a));
  printf("size of long = %d bytes\n", sizeof(b));
  printf("size of long long = %d bytes\n", sizeof(c));
  printf("size of long double= %d bytes\n", sizeof(d));
  return 0;
}
```

signed and unsigned

In C, signed and unsigned are type modifiers. You can alter the data storage of a data type by using them. For example,

unsigned int x;

int y;

Here, the variable x can hold only zero and positive values because we have used the unsigned modifier.

Considering the size of int is 4 bytes, variable y can hold values from -231 to 231-1, whereas variable x can hold values from 0 to 232-1.

Other data types defined in C programming are:

- bool Type
- Enumerated type
- Complex types

Constants

- Constants are like a variable, except that their value never changes during execution once defined.
- C Constants is the most fundamental and essential part of the C programming language. Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

- Constants are also called literals.
- Constants can be any of the data types.

It is considered best practice to define constants using only upper-case names.

const type constant_name;

const keyword defines a constant in C.

```
#include<stdio.h>
main()
{
  const int SIDE = 10;
  int area;
  area = SIDE*SIDE;
  printf("The area of the square with side: %d is: %d sq. units"
  , SIDE, area);
}
```

Putting const either before or after the type is possible.

int const SIDE = 10;

or

const int SIDE = 10;

Constants are categorized into two basic types, and each of these types has its subtypes/categories. These are:

Primary Constants

- Numeric Constants
    - Integer Constants
    - Real Constants
- **Character Constants**
    - Single Character Constants
    - String Constants
    - Backslash Character Constants

It's referring to a sequence of digits. Integers are of three types viz:

1. Decimal Integer
2. Octal Integer
3. Hexadecimal Integer

Example

15, -265, 0, 99818, +25, 045, 0X6

Real Constants

The numbers containing fractional parts like 99.25 are called real or floating points constant.

Single Character Constants

It simply contains a single character enclosed within ' and ' (a pair of single quote). It is to be noted that the character '8' is not the same as 8. Character constants have a specific set of integer values known as ASCII values (American Standard Code for Information Interchange).

Example*:*

'X', '5', ';'

String Constants

These are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and blank spaces. It is again to be noted that "G" and 'G' are different - because "G" represents a string as it is enclosed within a pair of double quotes whereas 'G' represents a single character.

Example

"Hello!", "2015", "2+1"

Backslash Character constants

C supports some character constants having a backslash in front of it. The lists of backslash characters have a specific meaning which is known to the compiler. They are also termed as "Escape Sequence".

\t is used to give a tab

\n is used to give a new line

Constants	Meaning
\a	beep sound
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\'	single quote
\"	double quote
\\	backslash
\0	null

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in ICS" etc.

2 ways to define constant in C

There are two ways to define constant

- **const keyword**
- **#define preprocessor**

1) C const keyword

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

const float PI=3.14;

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

```
#include<stdio.h>
int main(){
   const float PI=3.14;
   printf("The value of PI is: %f",PI);
   return 0;
}
```

Output:

```
The value of PI is: 3.140000
```

If you try to change the the value of PI, it will render compile time error.

```
#include<stdio.h>
int main(){
const float PI=3.14;
PI=4.5;
printf("The value of PI is: %f",PI);
   return 0;
}
```

Output:

```
Compile Time Error: Cannot modify a const object
```

2) C #define preprocessor

The #define preprocessor is also used to define constant. We will learn about #define preprocessor directive later.

Using #define preprocessor directive: This directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:

#define identifierName value

- **identifierName**: It is the name given to constant.

- **value**: This refers to any value assigned to identifierName.

Example:

```
#include<stdio.h>
#define val 10
#define floatVal 4.5
#define charVal 'G'

int main()
{
   printf("Integer Constant: %d\n",val);
   printf("Floating point Constant: %.1f\n",floatVal);
   printf("Character Constant: %c\n",charVal);

   return 0;
}
```

Output:

```
Integer Constant: 10

Floating point Constant: 4.5

Character Constant: G
```

using a const keyword: Using const keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a const keyword.

Below program shows how to use const to declare costants of different data types:

```
#include <stdio.h>
int main()
{
   // int constant
   const int intVal = 10;

   // Real constant
   const float floatVal = 4.14;

   // char constant
   const char charVal = 'A';

   // string constant
   const char stringVal[10] = "ABC";
   printf("Integer constant:%d \n", intVal );
   printf("Floating point constant: %.2f\n", floatVal );
   printf("Character constant: %c\n", charVal );
   printf("String constant: %s\n", stringVal);

   return 0;
}
```

Output:

```
Integer constant: 10

Floating point constant: 4.14

Character constant: A

String constant: ABC
```

What are literals?

Literals are the constant values assigned to the constant variables. We can say that the literals represent the fixed values that cannot be modified. It also contains memory but does not have references as variables. For example, const int =10; is a constant integer expression in which 10 is an integer literal.

Types of literals

There are four types of literals that exist in C programming:

- Integer literal
- Float literal
- Character literal
- String literal

Integer literal

It is a numeric literal that represents only integer type values. It represents the value neither in fractional nor exponential part.

It can be specified in the following three ways:

Decimal number (base 10)

It is defined by representing the digits between 0 to 9. For example, 45, 67, etc.

Octal number (base 8)

It is defined as a number in which 0 is followed by digits such as 0,1,2,3,4,5,6,7. For example, 012, 034, 055, etc.

Hexadecimal number (base 16)

It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-z) or (A-Z)).

An integer literal is suffixed by following two sign qualifiers:

L or l: It is a size qualifier that specifies the size of the integer type as long.

U or u: It is a sign qualifier that represents the type of the integer as unsigned. An unsigned qualifier contains only positive values.

Note: The order of the qualifier is not considered, i.e., both lu and ul are the same.

Let's look at a simple example of integer literal.

```
#include <stdio.h>
int main()
{
   const int a=23;  // constant integer literal
   printf("Integer literal : %d", a);
   return 0;
}
```

Output

```
Integer literal : 23
```

Float literal

It is a literal that contains only floating-point values or real numbers. These real numbers contain the number of parts such as integer part, real part, exponential part, and fractional part. The floating-point literal must be specified either in decimal or in exponential form. Let's understand these forms in brief.

Decimal form

The decimal form must contain either decimal point, exponential part, or both. If it does not contain either of these, then the compiler will throw an error. The decimal notation can be prefixed either by '+' or '-' symbol that specifies the positive and negative numbers.

Examples of float literal in decimal form are:

1.2, +9.0, -4.5

Let's see a simple example of float literal in decimal form.

```
#include <stdio.h>
int main()
{
   const float a=4.5; // constant float literal
   const float b=5.6; // constant float literal
   float sum;
   sum=a+b;
   printf("%f", sum);
   return 0;
}
```

Output

```
10.100000
```

Exponential form

The exponential form is useful when we want to represent the number, which is having a big magnitude. It contains two parts, i.e., mantissa and exponent. For example, the number is 2340000000000, and it can be expressed as 2.34e12 in an exponential form.

Syntax of float literal in exponential form

[+/-] <Mantissa> <e/E> [+/-] <Exponent>

Examples of real literal in exponential notation are:

+1e23, -9e2, +2e-25

Rules for creating an exponential notation

The following are the rules for creating a float literal in exponential notation:

- In exponential notation, the mantissa can be specified either in decimal or fractional form.
- An exponent can be written in both uppercase and lowercase, i.e., e and E.
- We can use both the signs, i.e., positive and negative, before the mantissa and exponent.
- Spaces are not allowed

Character literal

A character literal contains a single character enclosed within single quotes. If multiple characters are assigned to the variable, then we need to create a character array. If we try to store more than one character in a variable, then the warning of a multi-character character constant will be generated. Let's observe this scenario through an example.

```
#include <stdio.h>
int main()
{
   const char c='ak';
   printf("%c",c);
   return 0;
}
```

In the above code, we have used two characters, i.e., 'ak', within single quotes. So, this statement will generate a warning as shown below.

Warning generated:

```
main.c:6:18: warning: multi-character character constant

     [-Wmultichar]

   const char c='ak';

main.c:6:18: warning: implicit conversion from 'int' to 'char'

     changes value from 24939 to 107 [-Wconstant-conversion]
```

```
    const char c='ak';
              ~ ^~~~
2 warnings generated.
? ./main
```

Representation of character literal

A character literal can be represented in the following ways:

It can be represented by specifying a single character within single quotes. For example, 'a', 'b', etc.

We can specify the escape sequence character within single quotes to represent a character literal. For example, '\n', '\a', '\b'.

We can also use the ASCII in integer to represent a character literal. For example, the ascii value of 65 is 'A'.

The octal and hexadecimal notation can be used as an escape sequence to represent a character literal. For example, '\023', '\0x12'.

String literal

A string literal represents multiple characters enclosed within double-quotes. It contains an additional character, i.e., '\0' (null character), which gets automatically inserted. This null character specifies the termination of the string. We can use the '+' symbol to concatenate two strings.

For example,

String1= "ICS";

String2= "family";

To concatenate the above two strings, we use '+' operator, as shown in the below statement:

"ICS " + "family"= ICS family

Note: If we represent a single character, i.e., 'b', then this character will occupy a single byte as it is a character literal. And, if we represent the character within double quotes "b" then it will occupy more bytes as it is a string literal.

Just a Revision

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:

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

Let's understand each token one by one.

Keywords in C

Keywords in C can be defined as the pre-defined or the reserved words having its own importance, and each keyword has its own functionality. Since keywords are the pre-defined words used by the compiler, so they cannot be used as the variable names. If the keywords are used as the variable names, it means that we are assigning a different meaning to the keyword, which is not allowed. C language supports 32 keywords given below:

auto	double	int	Struct

break	else	long	switch
case	enum	register	Typedef
char	extern	return	Union
const	float	short	Unsigned
continue	for	signed	Void
default	goto	sizeof	Volatile
do	if	static	While

Identifiers in C

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

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.

Strings in C

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Now, we describe the strings in different ways:

char a[10] = "ICS"; // The compiler allocates the 10 bytes to the 'a' array.

char a[] = "ICS"; // The compiler allocates the memory at the run time.

char a[10] = {‘I’, ‘C’, ‘S’,'\0'}; // String is represented in the form of characters.

Operators in C

Operators in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

Unary Operator

A unary operator is an operator applied to the single operand. For example: increment operator (++), decrement operator (--), sizeof, (type)*.

Binary Operator

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

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

Constants in C

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.

There are two ways of declaring constant:

- Using const keyword
- Using #define pre-processor

Types of constants in C

Constant	Example
Integer constant	10, 11, 34, etc.
Floating-point constant	45.6, 67.8, 11.2, etc.
Octal constant	011, 088, 022, etc.
Hexadecimal constant	0x1a, 0x4b, 0x6b, etc.
Character constant	'a', 'b', 'c', etc.
String constant	"java", "c++", ".net", etc.

Special characters in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- Square brackets [ ]: The opening and closing brackets represent the single and multidimensional subscripts.
- Simple brackets ( ): It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- Curly braces { }: It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- Comma (,): It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.

- Hash/pre-processor (#): It is used for pre-processor directive. It basically denotes that we are using the header file.
- Asterisk (*): This symbol is used to represent pointers and also used as an operator for multiplication.
- Tilde (~): It is used as a destructor to free memory.
- Period (.): It is used to access a member of a structure or a union.

C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.

In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., stdbool.h. If we do not use the header file, then the program will not compile.

Syntax

```
bool variable_name;
```

In the above syntax, bool is the data type of the variable, and variable_name is the name of the variable.

Let's understand through an example.

```
#include <stdio.h>
#include<stdbool.h>
int main()
{
bool x=false; // variable initialization.
if(x==true) // conditional statements
{
printf("The value of x is true");
}
else
printf("The value of x is FALSE");
return 0;
}
```

In the above code, we have used <stdbool.h> header file so that we can use the bool type variable in our program. After the declaration of the header file, we create the bool type variable 'x' and assigns a 'false' value to it. Then, we add the conditional statements, i.e., if..else, to determine whether the value of 'x' is true or not.

Output

```
The value of x is FALSE
```

Boolean Array

Now, we create a bool type array. The Boolean array can contain either true or false value, and the values of the array can be accessed with the help of indexing.

Let's understand this scenario through an example.

```
#include <stdio.h>
#include<stdbool.h>
int main()
{
bool b[2]={true,false}; // Boolean type array
for(int i=0;i<2;i++) // for loop
{
printf("%d,",b[i]); // printf statement
}
return 0;
}
```

In the above code, we have declared a Boolean type array containing two values, i.e., true and false.

Output

```
1,0,
```

typedef

There is another way of using Boolean value, i.e., typedef. Basically, typedef is a keyword in C language, which is used to assign the name to the already existing datatype.

Let's see a simple example of typedef.

```
#include <stdio.h>
typedef enum{false,true} b;
int main()
```

```
{
b x=false; // variable initialization
if(x==true) // conditional statements
{
printf("The value of x is true");
}
else
{
printf("The value of x is false");
}
return 0;
}
```

In the above code, we use the Boolean values, i.e., true and false, but we have not used the bool type. We use the Boolean values by creating a new name of the 'bool' type. In order to achieve this, the typedef keyword is used in the program.

typedef enum{false,true} b;

The above statement creates a new name for the 'bool' type, i.e., 'b' as 'b' can contain either true or false value. We use the 'b' type in our program and create the 'x' variable of type 'b'.

Output

```
The value of x is false
```

Boolean with Logical Operators

The Boolean type value is associated with logical operators. There are three types of logical operators in the C language:

- &&(AND Operator): It is a logical operator that takes two operands. If the value of both the operands are true, then this operator returns true otherwise false
- ||(OR Operator): It is a logical operator that takes two operands. If the value of both the operands is false, then it returns false otherwise true.
- !(NOT Operator): It is a NOT operator that takes one operand. If the value of the operand is false, then it returns true, and if the value of the operand is true, then it returns false.

Let's understand through an example.

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

```
#include<stdbool.h>
int main()
{
bool x=false;
bool y=true;
printf("The value of x&&y is %d", x&&y);
printf("\nThe value of x||y is %d", x||y);
printf("\nThe value of !x is %d", !x);
}
```

Output

```
The value of x&&y is 0

The value of x||y is 1

The value of !x is 1
```

Stream in C Programming

Writing to and reading from some of the common devices, viz. floppy drives, CD-ROM drives, hard drives, network communication etc. Each of these devices have different characteristics and operating protocols. It’s the operating system that takes care of details of communication with these devices and provides with simpler and more uniform I/O interface to the programmer.

ANSI C abstracts all I/O as stream of bytes moving into and out of a C program. This stream of bytes is called **STREAM**. A C program is concerned only with creating the correct stream of bytes and interpreting the stream of bytes coming into the program as input. Mostly all I/O streams are fully buffered. This means that reading and writing data is actually copying data out of and into an area of memory called **buffer**. Because reading from and writing to memory is fast, this enhances the performance. Buffer for output stream is flushed, meaning physically written to device or file when it’s full. Writing full buffer is more efficient than writing data in small bits or pieces as program produces them. Similarly, input buffers are re-filled when they become empty by reading next large chunk of input from the device or file into the buffer.

The simple best strategy to debugging a C program that aborts abnormally is to use printf() statements throughout the program to identify the specific area where error occurred. Because of fully buffered I/O, the outputs of the calls to printf() are buffered and not showed up to the user when program aborts. This causes confusion to the user. In order to show up the outputs immediately before program aborts follow calls to **fflush() with call to printf().** For ex.

```
/* fflush.c -- program explores fflush() */

#include <stdio.h>
#include <stdlib.h>
#define DEBUG fprintf(fp, "values: a = %d and b = %d\n", a, b)
#define BSIZE 50
#define SUM(a,b)   ((a) + (b))

int main(void)
{

    int a = 10, b = 20;
    int *pa = &a, *pb = &b;
    int *pi = (int *)1000;

    FILE *fp;

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

    if (fp == NULL) {
        perror("File Open");
        exit(EXIT_FAILURE);
    }

    /* Program calls fflush() with DEBUG statements */
    /* fflush() forces the output buffer to be physically written */
    /* whether or not it's full */

    DEBUG;
    fflush(fp);

    fprintf(fp, "Sum of %d and %d is %d\n", a, b, SUM(a,b));
    fflush(fp);

    /* fushing the stdin is undefined */
    fflush(stdin);

    /* NULL to fflush() flushes all the output buffers */
    fflush(NULL);

    fprintf(fp, "What value pointer *pi is pointing at, %d\n", *pi);
    fflush(fp);

    printf("Add. of a and b is %p and %p\n", pa, pb);
    DEBUG;

    return 0;
}
```

Output

```
Segmentation fault (core dumped)

while output directed to file "fflush_out.txt" contains

values: a = 10 and b = 20

Sum of 10 and 20 is 30
```

Notice that above program aborts because of seg fault. But calls to fflush() with DEBUG statements causes the buffer to be physically written to the file identified by "fp" wherein DEBUG statements might help you trace the cause of abnormal program termination.

C - Input and Output

Input means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

Output means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

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

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

The file pointers are the means to access the file for reading and writing purpose.

The getchar() and putchar() Functions

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

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

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

  int c;

  printf( "Enter a value :");
  c = getchar( );

  printf( "\nYou entered: ");
  putchar( c );

  return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

```
Enter a value : this is test

You entered: t
```

The gets() and puts() Functions

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

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

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

```
#include <stdio.h>
int main( ) {
  char str[100];

  printf( "Enter a value :");
  gets( str );

  printf( "\nYou entered: ");
  puts( str );

  return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows −

```
Enter a value : this is test

You entered: this is test
```

The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream stdin and scans that input according to the format provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream stdout and produces the output according to the format provided.

The format can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better −

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

  char str[100];
  int i;

  printf( "Enter a value :");
  scanf("%s %d", str, &i);

  printf( "\nYou entered: %s %d ", str, i);
```

```
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows −

```
Enter a value : seven 7

You entered: seven 7
```

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

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. Input data can be entered from a standard input device by means of the C library function scanf. This function can be used to enter any combination of numeric values, single characters and strings. The function returns the number of data items that have been entered successfully.

scanf("format string",argument_list);

or

scanf(control string, arg1, arg2,.........,argN)

where control string refers to a string containing certain required formatting information, and arg1,arg2,….,argN are arguments that represent the individual data items. (Actually the arguments represent pointers that indicate the addresses of the data items within the computer’s memory. We will discuss pointers in greater detail in a future discussion, but until then it would be helpful to remember the fact that the arguments in the scanf function actually represent the addresses of the data items being entered.)

The control string consists of the individual group of characters called format specifiers, with one character group for each input data item. Each character group must begin with a per cent sign (%) and be followed by a conversion character which indicates the type of the data item. Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace characters (ie, white spaces, tabs or newline characters).

The arguments to a scanf function are written as variables or arrays whose types match the corresponding character groups in the control string. Each variable name must be preceded by an ampersand (&). However, character array names do not begin with an ampersand. The actual values of the arguments must correspond to the arguments in the scanf function in number, type and order.

If two or more data items are entered, they must be separated by whitespace characters. The data items may continue onto two or more lines, since the newline character is considered to be a whitespace character and can therefore separate consecutive data items

Program to print cube of 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.

Program to print sum of 2 numbers

```
#include<stdio.h>
int main(){
int x=0,y=0,result=0;

printf("enter first number:");
scanf("%d",&x);
printf("enter second number:");
scanf("%d",&y);

result=x+y;
printf("sum of 2 numbers:%d ",result);

return 0;
}
```

Output

```
enter first number:9

enter second number:9

sum of 2 numbers:18
```

Example

```
#include<stdio.h>

int main()

{

char a[20];

int   i;

float b;

printf(" n Enter the value of a, i and b");

scanf("%s %d %f", a, &i, &b);
return 0;

}
```
The most commonly used conversion characters are listed below:

Conversion Character	Data type of input data
c	Character
d	decimal integer
e	floating point value
f	floating point value
g	floating point value
h	short integer
I	decimal, hexadecimal or octal integer
o	octal integer
x	hexadecimal integer
s	String
U	unsigned decimal integer
[. . .]	string which may include whitespace characters

The s-type conversion character applies to a string that is terminated by a whitespace character. Therefore a string that includes whitespace characters cannot be entered in this manner. To do so, there are two ways:

1. The s-type conversion character is replaced by a sequence of characters enclosed in square brackets, designated as [. . .]. Whitespace characters are also included in the string so that a string that contains whitespaces may be read.

When the program is executed, successive characters will continue to be read as long as each input character matches one of the characters enclosed in the square brackets. The order of the characters in the square brackets need not correspond to the order of the characters being entered. As soon as an input character is encountered that does not match one of the characters within the brackets, the scanf function will stop reading any more characters and will terminate the string. A null character will then automatically be added to the end of the string.

Example:

```
#include<stdio.h>

int main()

{

char line[80];

scanf(" %[  ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);

printf("%s", line);

return 0;

}
```

If the input data is:

READING A STRING WITH WHITE SPACES

then the entire data will be assigned to the array line. However, if the input is:

Reading a String With white Spaces

then only the letters in uppercase (R, A, S, W, S) will be assigned to line, as all the characters in the control string are in uppercase.

2. To enter a string that includes whitespaces as well as uppercase and lowercase characters we can use the circumflex, ie (^), followed by a newline character within the brackets.

Example:

scanf("[^n]", line);

The circumflex causes the subsequent characters within the square brackets to be interpreted in the opposite manner. Thus, when the program is executed, characters will be read as long as a newline character is not encountered.

```
#include<stdio.h>

int main()

{
```

```
char line[80];

scanf(" %[  ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);

printf("%s", line);

return 0;

}
```

Reading Numbers: Specifying Field Width

The consecutive non-whitespace characters that define a data item collectively define a field. To limit the number of such characters for a data item, an unsigned integer indicating the field width precedes the conversion character. The input data may contain fewer characters than the specified field width. Extra characters will be ignored.

Example: If a and b are two integer variables and the following statement is being used to read their values:

```
scanf( "%3d %3d", &a, &b);
```

and if the input data is: 1 4

then a will be assigned 1 and b 4.

If the input data is 123 456 then a=123 and b=456.

If the input is 1234567, then a=123 and b=456. 7 is ignored.

If the input is 123 4 56 (space inserted by a typing mistake), then a=123 and b=4. This is because the space acts as a data item separator.

Assignment Suppression

The % sign is followed by an asterisk (*) to skip over a data item without assigning it to the designated variable or array.

Example:

*scanf("%s %*d %f", a, &i,&b);*

If the input is: Alice 12 34.5 then a=Alice and b= 34.5. However, 12 will not be assigned to i because of the asterisk which is interpreted as an assignment suppression character.

%[flags][width][.precision][length]specifier

Where the specifier character at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed.	

The corresponding argument must be a pointer to a signed int.

The number of characters written so far is stored in the pointed location.

%	A % followed by another % character will write a single % to the stream. %

The format specifier can also contain sub-specifiers: flags, width, .precision and modifiers (in that order), which are optional and follow these specifications:

Flags	description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see width sub-specifier).

Width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed.

	For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

scanf()

The C library function int scanf(const char *format, ...) reads formatted input from stdin.

Declaration

Following is the declaration for scanf() function.

*int scanf(const char *format, ...)*

Parameters

format − This is the C string that contains one or more of the following items −

Whitespace character, Non-whitespace character and Format specifiers. A format specifier will be like [=%[*][width][modifiers]type=] as explained below −

Argument	Description
*	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
Width	This specifies the maximum number of characters to be read in the current reading operation.
Modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)

Type	A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers

type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

additional arguments − Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

On success, the function returns the number of items of the argument list successfully read. If a reading error happens or the end-of-file is reached while reading, the proper indicator is set (feof or ferror) and, if either happens before any data could be successfully read, EOF is returned.

Example

The following example shows the usage of scanf() function.

```
#include <stdio.h>

int main () {
   char str1[20], str2[30];

   printf("Enter name: ");
   scanf("%s", str1);

   printf("Enter your website name: ");
   scanf("%s", str2);

   printf("Entered Name: %s\n", str1);
   printf("Entered Website:%s", str2);

   return(0);
}
```

Output

```
Enter name: admin

Enter your website name: www.ICS.com

Entered Name: admin

Entered Website: www.ics.com
```

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

An operator is a symbol that operates on a value or a variable. For example: + is an operator to perform addition.

C has a wide range of operators to perform various operations.

C Arithmetic Operators

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

Operator	**Meaning of Operator**
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```
// 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;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
```

Remainder when a divided by b=1

The operators +, - and * computes addition, subtraction, and multiplication respectively as you might have expected.

In normal calculation, 9/4 = 2.25. However, the output is 2 in the program.

It is because both the variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a=9 is divided by b=4, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5

// Both operands are integers

c/d = 2

C Increment and Decrement Operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators

#include <stdio.h>
int main()
{
   int a = 10, b = 100;
   float c = 10.5, d = 100.5;

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

   return 0;
}
```

Output

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

Here, the operators ++ and -- are used as prefixes. These two operators can also be used as postfixes like a++ and a--. Visit this page to learn more about how increment and decrement operators work when used as postfix.

C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b

Operator	Example	Same as
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include <stdio.h>
int main()
{
   int a = 5, c;

   c = a;      // c is 5
   printf("c = %d\n", c);
   c += a;     // c is 10
   printf("c = %d\n", c);
   c -= a;     // c is 5
   printf("c = %d\n", c);
   c *= a;     // c is 25
   printf("c = %d\n", c);
   c /= a;     // c is 5
   printf("c = %d\n", c);
   c %= a;     // c = 0
   printf("c = %d\n", c);

   return 0;
}
```

Output

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

c = 5
c = 0

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	**Meaning of Operator**	**Example**
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10;

   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
   printf("%d < %d is %d \n", a, b, a < b);
   printf("%d < %d is %d \n", a, c, a < c);
   printf("%d != %d is %d \n", a, b, a != b);
   printf("%d != %d is %d \n", a, c, a != c);
```

```
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

Output

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

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the	If c = 5 then, expression !(c==5) equals to 0.

Operator	Meaning	Example
	operand is 0	

Example 5: Logical Operators

```
// Working of logical operators

#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
   printf("(a == b) && (c > b) is %d \n", result);

   result = (a == b) && (c < b);
   printf("(a == b) && (c < b) is %d \n", result);

   result = (a == b) || (c < b);
   printf("(a == b) || (c < b) is %d \n", result);

   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) is %d \n", result);

   result = !(a != b);
   printf("!(a != b) is %d \n", result);

   result = !(a == b);
   printf("!(a == b) is %d \n", result);

   return 0;
}
```

Output

```
•  (a == b) && (c > b) is 1
•  (a == b) && (c < b) is 0
•  (a == b) || (c < b) is 1
•  (a != b) || (c < b) is 0
•  !(a != b) is 1
•  !(a == b) is 0
```

Explanation of logical operator program

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	**Meaning of operators**
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example : sizeof Operator

```
#include <stdio.h>
int main()
{
   int a;
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));

   return 0;
}
```

Output

```
Size of int = 4 bytes

Size of float = 4 bytes

Size of double = 8 bytes

Size of char = 1 byte
```

Other operators such as ternary operator ?:, reference operator &, dereference operator * and member selection operator -> will be discussed in later.

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:

```
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