The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

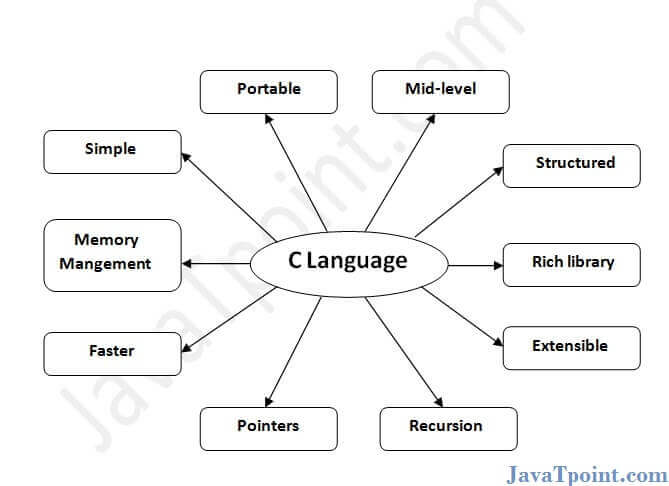
C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

**C programming language** was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Features of C Language



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

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

## **1) Simple**

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

## **2) Machine Independent or Portable**

Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language.

## **3) Mid-level programming language**

Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

## **4) Structured programming language**

C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

## **5) Rich Library**

C **provides a lot of inbuilt functions** that make the development fast.

## **6) Memory Management**

It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

**7) Speed**

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

## **8) Pointer**

C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

## **9) Recursion**

In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

## **10) Extensible**

C language is extensible because it **can easily adopt new features**.

First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

1. #include <stdio.h>
2. **int** main(){
3. printf("Hello C Language");
4. **return** 0;
5. }

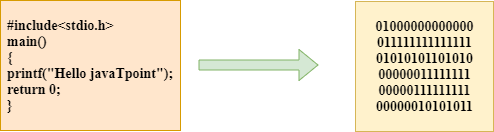
**#include <stdio.h>** includes the **standard input output** library functions. The printf() function is defined in stdio.h . **int main()** The **main() function is the entry point of every program** in c language.

**printf()** The printf() function is **used to print data** on the console.

**return 0** The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

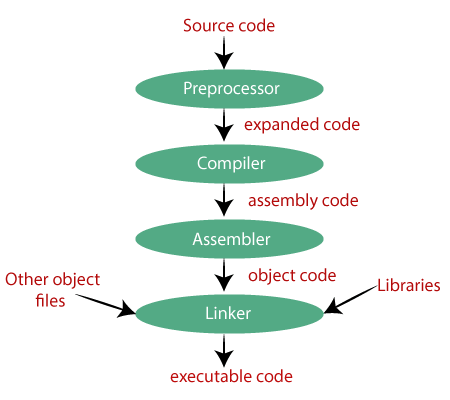
## **What is a compilation?**

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.



The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if **<stdio.h>,** the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the **'stdio.h'** file.



### **Preprocessor**

### The source code is the code which is written in a text editor. This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

### **Compiler**

### The compiler converts this code into assembly code.

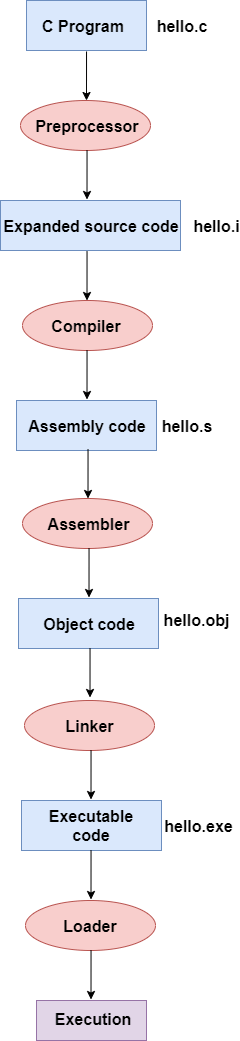
### **Assembler**

### The assembly code is converted into object code by using an assembler.

If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

**Linker**

we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files.



* Firstly, the input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i.**
* The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s.**
* This assembly code is then sent to the assembler, which converts the assembly code into object code.
* After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

# printf() and scanf() in C

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

### **printf() function**

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

The syntax of printf() function is given below:

1. 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.

1. scanf("format string",argument\_list);

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.

# Variables in C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified. The example of declaring the variable is given below:

1. **int** a;
2. **float** b;
3. **char** c;

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

;//declaring 2 variable of integer type

1. **int** a=10, b=20
2. **float** f=20.8;
3. **char** c='A';

## **Rules for defining variables**

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

**Valid variable names:**

1. **int** a; 2. **int** \_ab;   3. **int** a30;

**Invalid variable names:**

1. **int** 2;   2. **int** a b;   3. **int** **long**;

## **Types of Variables in C**

There are many types of variables in c:

### **Local Variable**

A variable that is declared inside the function or block is called a local variable.

It must be declared at the start of the block.

1. **void** function1(){
2. **int** x=10;//local variable
3. }

You must have to initialize the local variable before it is used.

### **Global Variable**

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.

It must be declared at the start of the block.

1. **int** value=20;//global variable
2. **void** function1(){
3. **int** x=10;//local variable
4. }

### **Static Variable**

A variable that is declared with the static keyword is called static variable. It retains its value between multiple function calls.

1. **void** function1(){
2. **int** x=10;//local variable
3. **static** **int** y=10;//static variable
4. x=x+1;
5. y=y+1;
6. printf("%d,%d",x,y);
7. }

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

### **Automatic Variable**

### All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

1. **void** main(){
2. **int** x=10;//local variable (also automatic)
3. auto **int** y=20;//automatic variable
4. }

### **External Variable**

### We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

*myfile.h*

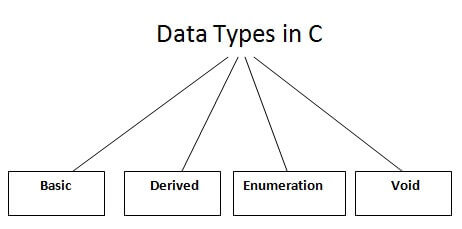
1. **extern** **int** x=10;//external variable (also global)

*program1.c*

1. #include "myfile.h"
2. #include <stdio.h>
3. **void** printValue(){
4. printf("Global variable: %d", global\_variable);
5. }

# Data Types in C

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



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 |

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 |  |

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

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

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

# C Identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore.

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., '+')

## **Differences between Keyword and Identifier**

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

**Let's understand through an example.**

1. **int** main()  {
2. **int** a=10;
3. **int** A=20;
4. printf("Value of a is : %d",a);
5. printf("\nValue of A is :%d",A);
6. **return** 0;
7. }

**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.

# 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 [ ( == ) , ( >,>= ) , (< ,<= ), ( ! = ) ] defines relation b/w two operators.
* Shift Operators [ << >> ]
* Logical Operators [ && , || , ! ]
* Bitwise Operators [Bitwise and & , Bitwise or | , Bitwise xor ^]
* Ternary or Conditional Operators
* Assignment Operator [ =, += , -= , \*= , /= ,%= ,>>= , <<= , &= ,^=, |= ]
* Misc Operator

## **Precedence of Operators in C**

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

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

1. **int** value=10+20\*10;

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

The precedence and associativity of C operators is given below:

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

# Comments in C

There are 2 types of comments in the C language.

1. Single Line Comments  represented by double slash \\.
2. Multi-Line Comments represented by slash asterisk \\* ... \*\.

# C Format Specifier

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

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

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

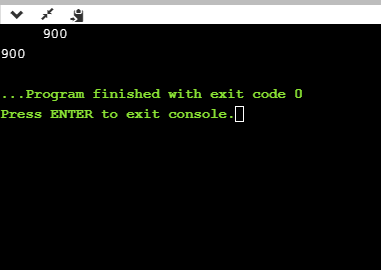
## **Minimum Field Width Specifier**

Suppose we want to display an output that occupies a minimum number of spaces on the screen. You can achieve this by displaying an integer number after the percent sign of the format specifier.

1. **int** main()
2. {
3. **int** x=900;
4. printf("%8d", x);
5. printf("\n%-8d",x);
6. **return** 0;
7. }

In the above program, %8d specifier displays the value after 8 spaces while %-8d specifier will make a value left-aligned.

**Output**



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Escape Sequence in C An escape sequence in C language is a sequence of characters that doesn't represent itself  When used inside string literal or character.  It is composed of two or more characters starting with backslash \. For example: \n represents  new line. **List of Escape Sequences in C**  |  |  | | --- | --- | | **Escape Sequence** | **Meaning** | | \a | Alarm or Beep | | \b | Backspace | | \f | Form Feed | | \n | New Line | | \r | Carriage Return | | \t | Tab (Horizontal) | | \v | Vertical Tab | | \\ | Backslash | | \' | Single Quote | | \" | Double Quote | | \? | Question Mark | | \nnn | octal number | | \xhh | hexadecimal number | | \0 | Null | |

## **Escape Sequence Example**

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\"");
5. **return** 0;
6. }

**Output:**

You

are

learning

'c' language

"Do you know C language"

## **What is ASCII code?**

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

In [C programming language](https://www.javatpoint.com/c-programming-language-tutorial), a character variable does not contain a character value itself rather the ascii value of the character variable. The ascii value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127. For example, the ascii value of 'A' is 65.

## Create a program which will display the ascii value of all the characters.

1. #include <stdio.h>

**int** main()

{

**int** k;   // variable declaration

**for**(**int** k=0;k<=255;k++)  // for loop from 0-255

 {

     printf("\nThe ascii value of %c is %d", k,k);

 }

return 0;

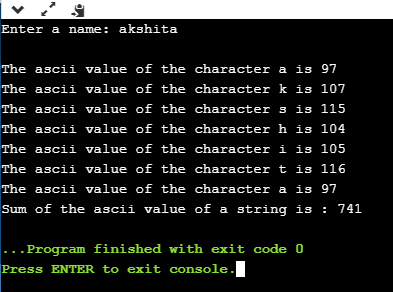
}

# create the program which will sum the ascii value of a string.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** sum=0;  // variable initialization
5. **char** name[20];  // variable initialization
6. **int** i=0;  // variable initialization
7. printf("Enter a name: ");
8. scanf("%s", name);
9. **while**(name[i]!='\0')  // while loop
10. {
11. printf("\nThe ascii value of the character %c is %d", name[i],name[i]);
12. sum=sum+name[i];
13. i++;
14. }
15. printf("\nSum of the ascii value of a string is : %d", sum);
16. **return** 0;
17. }

In the above code, we are taking user input as a string. After taking user input, we execute the **while** loop which adds the ascii value of all the characters of a string and stores it in a '**sum**' variable.

**Output**



**Constants in C**

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

There are different types of constants in C programming.

## **List of Constants in C**

|  |  |
| --- | --- |
| **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 javatpoint" etc. |

## **2 ways to define constant in C**

1. const keyword
2. #define preprocessor

## **1) C const keyword**

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

1. **const** **float** PI=3.14;

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

1. #include<stdio.h>
2. **int** main(){
3. **const** **float** PI=3.14;
4. printf("The value of PI is: %f",PI);
5. **return** 0;
6. }

**Output:**

The value of PI is: 3.140000

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

1. #include<stdio.h>
2. **int** main(){
3. **const** **float** PI=3.14;
4. PI=4.5;
5. printf("The value of PI is: %f",PI);
6. **return** 0;
7. }

**Output:**

Compile Time Error: Cannot modify a const object

**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**](https://www.javatpoint.com/c-programming-language-tutorial)**:**

* **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)**

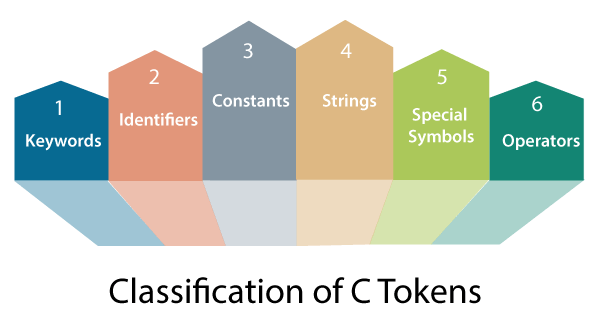
It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-z) or (A-Z)).

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](https://www.javatpoint.com/c-programs).

**Classification of tokens in C**

Tokens in [C language](https://www.javatpoint.com/c-programming-language-tutorial) can be divided into the following categories:



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

# Keywords in C

|  |  |  |  |
| --- | --- | --- | --- |
| **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** |

* [Keywords in C](https://www.javatpoint.com/keywords-in-c) can be defined as the **pre-defined** orthe **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:

Identifiers in C

[Identifiers in C](https://www.javatpoint.com/c-identifiers) 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.

# Strings in C

[Strings in C](https://www.javatpoint.com/c-strings) 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. strings in different ways:

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

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

char a[10] = {'j','a','v','a','t','p','o','i','n','t','\0'}; // String is represented in the form of characters.

# Operators in C

[Operators in C](https://www.javatpoint.com/c-operators) 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**](https://www.javatpoint.com/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++](https://www.javatpoint.com/cpp-tutorial), 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**

1. **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.**

1. #include <stdio.h>
2. #include<stdbool.h>
3. **int** main()
4. {
5. **bool** x=**false**; // variable initialization.
6. **if**(x==**true**) // conditional statements
7. {
8. printf("The value of x is true");
9. }
10. **else**
11. printf("The value of x is FALSE");
12. **return** 0;
13. }

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.

**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. example

1. #include <stdio.h>
2. #include<stdbool.h>
3. **int** main()

{

1. **bool** b[2]={**true**,**false**}; // Boolean type array
2. **for**(**int** i=0;i<2;i++) // for loop
3. {
4. printf("%d,",b[i]); // printf statement
5. }
6. **return** 0;
7. }

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](https://www.javatpoint.com/keywords-in-c), which is used to assign the name to the already existing datatype.

**Let's see a simple example of typedef.**

1. #include <stdio.h>
2. **typedef** **enum**{**false**,**true**} b;
3. **int** main()  {
4. b x=**false**; // variable initialization
5. **if**(x==**true**)  {   // conditional statements
6. printf("The value of x is true");
7. }  **else**  {
8. printf("The value of x is false");
9. }
10. **return** 0;
11. }

## **Boolean with Logical Operators**

The Boolean type value is associated with logical operators. There are three types of logical operators in the [C language](https://www.javatpoint.com/c-programming-language-tutorial):

**&&(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.**

1. #include <stdio.h>
2. #include<stdbool.h>
3. **int** main()
4. {
5. **bool** x=**false**;
6. **bool** y=**true**;
7. printf("The value of x&&y is %d", x&&y);
8. printf("\nThe value of x||y is %d", x||y);
9. printf("\nThe value of !x is %d", !x);
10. }

**Output**

The value of x&&y is 0

The value of x||y is 1

The value of !x is 1

# Static in C

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

### **Static keyword can be used in the following situations:**

# ****Static global variable****

When a global variable is declared with a static keyword, then it is known as a static global variable. It is declared at the top of the program, and its visibility is throughout the program.

**Static function**

When a function is declared with a static keyword known as a static function. Its lifetime is throughout the program.

* **Static local variable**

When a local variable is declared with a static keyword, then it is known as a static local variable. The memory of a static local variable is valid throughout the program, but the scope of visibility of a variable is the same as the automatic local variables. However, when the function modifies the static local variable during the first function call, then this modified value will be available for the next function call also.

* **Static member variable**

When the member variables are declared with a static keyword in a class, then it is known as static member variables. They can be accessed by all the instances of a class, not with a specific instance.

* **Static method**

The member function of a class declared with a static keyword is known as a static method. It is accessible by all the instances of a class, not with a specific instance.

### **Static variable**

A static variable is a variable that persists its value across the various function calls.

**Syntax**

The syntax of a static variable is given below:

1. **static** data\_type variable\_name;

**Let's look at a simple example of static variable.**

1. #include <stdio.h>
2. **int** main()
3. {
4. printf("%d",func());
5. printf("\n%d",func());
7. **return** 0;
8. }
9. **int** func()
10. {
11. **static** **int** count=0;
12. count++;
13. **return** count;
14. }

In the above code, we have declared the count variable as static. When the func() is called, the value of count gets updated to 1, and during the next function call, the value of the count variable becomes 2. Therefore, we can say that the value of the static variable persists within the function call.

**Output**

1

2

### **Static Function**

As we know that non-static functions are global by default means that the function can be accessed outside the file also, but if we declare the function as static, then it limits the function scope. The static function can be accessed within a file only.

**The static function would look like as:**

1. **static** **void** func()
2. {
3. printf("Hello javaTpoint");
4. }

**Let's understand through an example.**

1. #include <stdio.h>
2. **int** main()
3. {
4. printf("%d",func());
5. printf("\n%d",func());
6. **return** 0;
7. }
8. **int** func()
9. {
10. **int** count=0; // variable initialization
11. count++; // incrementing counter variable
13. **return** count; }

# Programming Errors in C

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

**There are mainly five types of errors exist in C programming:**

* **Syntax error**
* **Run-time error**
* **Linker error**
* **Logical error**
* **Semantic error**

### **Syntax error**

### Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected. Commonly occurred syntax errors are:

* If we miss the parenthesis ( } ) while writing the code.
* Displaying the value of a variable without its declaration.
* If we miss the semicolon ( ; ) at the end of the statement.

### **Run-time error**

### Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

### **Linker error**

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main().**

### **Logical error**

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors.

#include <stdio.h>

**int** main()

{

**int** sum=0; // variable initialization

**int** k=1;

**for**(**int** i=1;i<=10;i++); // logical error, as we put the semicolon after loop

   {

       sum=sum+k;

       k++;

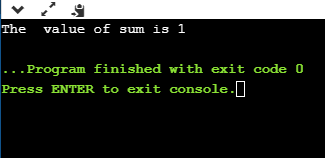
   }

printf("The  value of sum is %d", sum);

**return** 0;

}

**Output**



In the above code, we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

### **Compile-time errors**

Compile-time errors are the errors that occurred when we write the wrong syntax. If we write the wrong syntax or semantics of any programming language, then the compile-time errors will be thrown by the compiler. The compiler will not allow to run the program until all the errors are removed from the program. When all the errors are removed from the program, then the compiler will generate the executable file.

The compile-time errors can be:

* Syntax errors
* Semantic errors

# Conditional Operator in C

The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

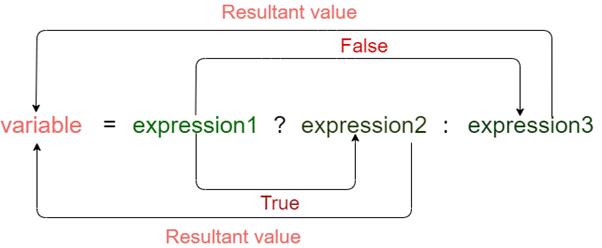
As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the '[if-else](https://www.javatpoint.com/c-if-else)' statement as 'if-else' statement is also a decision-making statement.

### **Syntax of a conditional operator**

1. Expression1? expression2: expression3;

**The pictorial representation of the above syntax is shown below:**



**Meaning of the above syntax.**

* In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
* If the expression1 results into a true value, then the expression2 will execute.
* The expression2 is said to be true only when it returns a non-zero value.
* If the expression1 returns false value then the expression3 will execute.
* The expression3 is said to be false only when it returns zero value.

**Let's understand the ternary or conditional operator through an example.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** age;  // variable declaration
5. printf("Enter your age");
6. scanf("%d",&age);   // taking user input for age variable
7. (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting"));
8. // conditional operator
9. **return** 0;
10. }

How a conditional operator is used to assign the value to a variable?

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=5,b;  // variable declaration
5. b=((a==5)?(3):(2)); // conditional operator
6. printf("The value of 'b' variable is : %d",b);
7. **return** 0;
8. }

we have declared two variables, i.e., 'a' and 'b', and assign 5 value to the 'a' variable. After the declaration, we are assigning value to the 'b' variable by using the conditional operator. If the value of 'a' is equal to 5 then 'b' is assigned with a 3 value otherwise 2.

The above output shows that the value of 'b' variable is 3 because the value of 'a' variable is equal to 5.

# As we know that the behavior of conditional operator and 'if-else' is similar but they have some differences. Let's look at their differences.

* A conditional operator is a single programming statement, while the 'if-else' statement is a programming block in which statements come under the parenthesis.
* A conditional operator can also be used for assigning a value to the variable, whereas the 'if-else' statement cannot be used for the assignment purpose.
* It is not useful for executing the statements when the statements are multiple, whereas the 'if-else' statement proves more suitable when executing multiple statements.
* The nested ternary operator is more complex and cannot be easily debugged, while the nested 'if-else' statement is easy to read and maintain.

# Bitwise Operator in C

The bitwise operators are the operators used to perform the operations on the data at the bit-level. When we perform the bitwise operations, then it is also known as bit-level programming. It consists of two digits, either 0 or 1. It is mainly used in numerical computations to make the calculations faster.

The following is the list of the bitwise operators:

|  |  |
| --- | --- |
| **Operator** | **Meaning of operator** |
| & | Bitwise AND operator |
| | | Bitwise OR operator |
| ^ | Bitwise exclusive OR operator |
| ~ | One's complement operator (unary operator) |
| << | Left shift operator |
| >> | Right shift operator |

**Let's look at the truth table of the bitwise operators.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **Y** | **X&Y** | **X|Y** | **X^Y** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

### **Bitwise AND operator**

Bitwise AND operator is denoted by the single ampersand sign (&). Two integer operands are written on both sides of the (&) operator. If the corresponding bits of both the operands are 1, then the output of the bitwise AND operation is 1; otherwise, the output would be 0.

For example,

1. We have two variables a and b.
2. a =6;
3. b=4;
4. The binary representation of the above two variables are given below:
5. a = 0110
6. b = 0100
7. When we apply the bitwise AND operation in the above two variables, i.e., a&b, the output would be:
8. Result = 0100

As we can observe from the above result that bits of both the variables are compared one by one. If the bit of both the variables is 1 then the output would be 1, otherwise 0.

## Let's understand the bitwise AND operator through the program.

## #include <stdio.h>

1. **int** main()
2. {
3. **int** a=6, b=14;  // variable declarations
4. printf("The output of the Bitwise AND operator a&b is %d",a&b);
5. **return** 0;
6. }

In the above code, we have created two variables, i.e., 'a' and 'b'. The values of 'a' and 'b' are 6 and 14 respectively. The binary value of 'a' and 'b' are 0110 and 1110, respectively. When we apply the AND operator between these two variables,

**a AND b = 0110 && 1110 = 0110**

**Output** The output of the Bitwise AND operator a&b is 6

### **Bitwise OR operator**

The bitwise OR operator is represented by a single vertical sign (|). Two integer operands are written on both sides of the (|) symbol. If the bit value of any of the operand is 1, then the output would be 1, otherwise 0.

For example,

1. We consider two variables,
2. a = 23;
3. b = 10;
4. The binary representation of the above two variables would be:
5. a = 0001 0111
6. b = 0000 1010
7. When we apply the bitwise OR operator in the above two variables, i.e., a|b , then the output would be:
8. Result = 0001 1111

As we can observe from the above result that the bits of both the operands are compared one by one; if the value of either bit is 1, then the output would be 1 otherwise 0.

**Let's understand the bitwise OR operator through a program.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=23,b=10;  // variable declarations
5. printf("The output of the Bitwise OR operator a|b is %d",a|b);
6. **return** 0;
7. }

**Output** The output of the Bitwise OR operator a|b is 6

### **Bitwise exclusive OR operator**

Bitwise exclusive OR operator is denoted by (^) symbol. Two operands are written on both sides of the exclusive OR operator. If the corresponding bit of any of the operand is 1 then the output would be 1, otherwise 0.

For example,

1. We consider two variables a and b,
2. a = 12;
3. b = 10;
4. The binary representation of the above two variables would be:
5. a = 0000 1100
6. b = 0000 1010
7. When we apply the bitwise exclusive OR operator in the above two variables (a^b), then the result would be:
8. Result = 0000 1110

As we can observe from the above result that the bits of both the operands are compared one by one; if the corresponding bit value of any of the operand is 1, then the output would be 1 otherwise 0.

**Let's understand the bitwise exclusive OR operator through a program.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=12,b=10;  // variable declarations
5. printf("The output of the Bitwise exclusive OR operator a^b is %d",a^b);
6. **return** 0;
7. }

**Output** The output of the Bitwise ZOR operator a^b is 6

### **Bitwise complement operator**

The bitwise complement operator is also known as one's complement operator. It is represented by the symbol tilde (~). It takes only one operand or variable and performs complement operation on an operand. When we apply the complement operation on any bits, then 0 becomes 1 and 1 becomes 0.

For example,

1. If we have a variable named 'a',
2. a = 8;
3. The binary representation of the above variable is given below:
4. a = 1000
5. When we apply the bitwise complement operator to the operand, then the output would be:
6. Result = 0111

As we can observe from the above result that if the bit is 1, then it gets changed to 0 else 1.

**Let's understand the complement operator through a program.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=8;  // variable declarations
5. printf("The output of the Bitwise complement operator ~a is %d",~a);
6. **return** 0;
7. }

**Output** The output of the Bitwise COMPLEMENT operator ~a is -9.

### **Bitwise shift operators**

Two types of bitwise shift operators exist in C programming. The bitwise shift operators will shift the bits either on the left-side or right-side. Therefore, we can say that the bitwise shift operator is divided into two categories:

* Left-shift operator
* Right-shift operator

**Left-shift operator**

It is an operator that shifts the number of bits to the left-side.

**Syntax of the left-shift operator is given below:**

1. Operand << n

**Where,**

**Operand is an integer expression on which we apply the left-shift operation.**

**n is the number of bits to be shifted.**

In the case of Left-shift operator, 'n' bits will be shifted on the left-side. The 'n' bits on the left side will be popped out, and 'n' bits on the right-side are filled with 0.

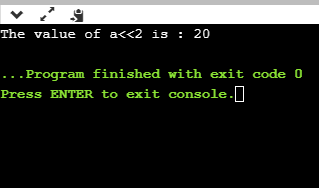
**For example,**

1. Suppose we have a statement:
2. **int** a = 5;
3. The binary representation of 'a' is given below:
4. a = 0101
5. If we want to left-shift the above representation by 2, then the statement would be:
6. a << 2;
7. 0101<<2 = 00010100

**Let's understand through a program.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=5; // variable initialization
5. printf("The value of a<<2 is : %d ", a<<2);
6. **return** 0;
7. }

**Output**



**Right-shift operator**

It is an operator that shifts the number of bits to the right side.

**Syntax of the right-shift operator is given below:**

1. Operand >> n;

**Where,**

Operand is an integer expression on which we apply the right-shift operation.

N is the number of bits to be shifted.

In the case of the right-shift operator, 'n' bits will be shifted on the right-side. The 'n' bits on the right-side will be popped out, and 'n' bits on the left-side are filled with 0.

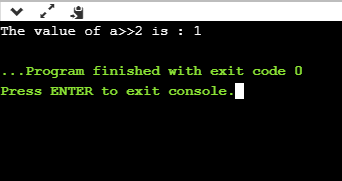
**For example,**

1. Suppose we have a statement,
2. **int** a = 7;
3. The binary representation of the above variable would be:
4. a = 0111
5. If we want to right-shift the above representation by 2, then the statement would be:
6. a>>2;
7. 0000 0111 >> 2 = 0000 0001

**Let's understand through a program.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** a=7; // variable initialization
5. printf("The value of a>>2 is : %d ", a>>2);
6. **return** 0;
7. }

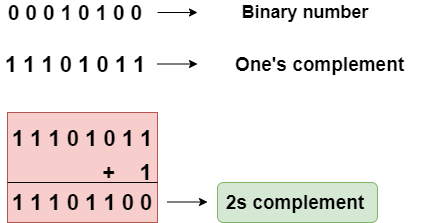
**Output**



# What is the 2s complement in C?

The 2s complement in C is generated from the 1s complement in C. As we know that the 1s complement of a binary number is created by transforming bit 1 to 0 and 0 to 1; the 2s complement of a binary number is generated by adding one to the 1s complement of a binary number.

In short, we can say that the 2s complement in C is defined as the sum of the one's complement in C and one.



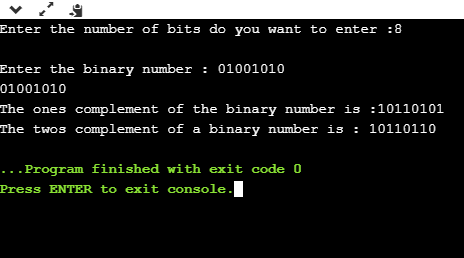
In the above figure, the binary number is equal to 00010100, and its one's complement is calculated by transforming the bit 1 to 0 and 0 to 1 vice versa. Therefore, one's complement becomes 11101011. After calculating one's complement, we calculate the two's complement by adding 1 to the one's complement, and its result is 11101100.

Let's create a program of 2s complement.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** n;  // variable declaration
5. printf("Enter the number of bits do you want to enter :");
6. scanf("%d",&n);
7. **char** binary[n+1];  // binary array declaration;
8. **char** onescomplement[n+1]; // onescomplement array declaration
9. **char** twoscomplement[n+1]; // twoscomplement array declaration
10. **int** carry=1; // variable initialization
11. printf("\nEnter the binary number : ");
12. scanf("%s", binary);
13. printf("%s", binary);
14. printf("\nThe ones complement of the binary number is :");
16. // Finding onescomplement in C
17. **for**(**int** i=0;i<n;i++)
18. {
19. **if**(binary[i]=='0')
20. onescomplement[i]='1';
21. **else** **if**(binary[i]=='1')
22. onescomplement[i]='0';
23. }
24. onescomplement[n]='\0';
25. printf("%s",onescomplement);

28. printf("\nThe twos complement of a binary number is : ");
30. // Finding twoscomplement in C
31. **for**(**int** i=n-1; i>=0; i--)
32. {
33. **if**(onescomplement[i] == '1' && carry == 1)
34. {
35. twoscomplement[i] = '0';
36. }
37. **else** **if**(onescomplement[i] == '0' && carry == 1)
38. {
39. twoscomplement[i] = '1';
40. carry = 0;
41. }
42. **else**
43. {
44. twoscomplement[i] = onescomplement[i];
45. }
46. }
47. twoscomplement[n]='\0';
48. printf("%s",twoscomplement);
49. **return** 0;
50. }

**Output**



### **Analysis of the above program,**

* First, we input the number of bits, and it gets stored in the '**n**' variable.
* After entering the number of bits, we declare character array, i.e., **char binary[n+1],** which holds the binary number. The '**n**' is the number of bits which we entered in the previous step; it basically defines the size of the array.
* We declare two more arrays, i.e., **onescomplement[n+1]**, and **twoscomplement[n+1].** The **onescomplement[n+1]** array holds the ones complement of a binary number while the **twoscomplement[n+1]** array holds the two's complement of a binary number.
* Initialize the **carry** variable and assign 1 value to this variable.
* After declarations, we input the binary number.
* Now, we simply calculate the one's complement of a binary number. To do this, we create a **loop** that iterates throughout the binary array, **for(int i=0;i<n;i++)**. In for loop, the condition is checked whether the bit is 1 or 0. If the bit is 1 then **onescomplement[i]=0**else **onescomplement[i]=1**. In this way, one's complement of a binary number is generated.
* After calculating one's complement, we generate the 2s complement of a binary number. To do this, we create a **loop** that iterates from the last element to the starting element. In for loop, we have three conditions:
  + If the bit of onescomplement[i] is 1 and the value of carry is 1 then we put 0 in twocomplement[i].
  + If the bit of onescomplement[i] is 0 and the value of carry is 1 then we put 1 in twoscomplement[i] and 0 in carry.
  + If the above two conditions are false, then onescomplement[i] is equal to twoscomplement[i].