

**C** is a procedural programming language initially developed by **Dennis Ritchie** in the year **1972** at Bell Laboratories of AT&T Labs. It was mainly developed as a system programming language to write the **UNIX operating system**.

# Structure of C Program

|      |   |                                 |           |
|------|---|---------------------------------|-----------|
|      | 1 | <i>#include &lt;stdio.h&gt;</i> | Header    |
|      | 2 | <i>int main(void)</i>           | Main      |
| BODY | 3 | <i>{</i>                        |           |
|      | 4 | <i>printf("Hello World");</i>   | Statement |
|      | 5 | <i>return 0;</i>                | Return    |
|      | 6 | <i>}</i>                        |           |

## 1. Header Files Inclusion – Line 1 [`#include <stdio.h>`]

The first and foremost component is the inclusion of the Header files in a C program. A header file is a file with extension `.h` which contains C function declarations and macro definitions to be shared between several source files. All lines that start with `#` are processed by a preprocessor which is a program invoked by the compiler. In the above example, the preprocessor copies the preprocessed code of `stdio.h` to our file. The `.h` files are called header files in C.

## 2. Main Method Declaration – Line 2 [int main()]

The next part of a C program is to declare the main() function. It is the entry point of a C program and the execution typically begins with the first line of the main(). The empty brackets indicate that the main doesn't take any parameter (See [this](#) for more details). The int that was written before the main indicates the return type of main(). The value returned by the main indicates the status of program termination. See [this](#) post for more details on the return type.

A token in C can be defined as the smallest individual element of the C programming language that is meaningful to the compiler. It is the basic component of a C program.

# Types of Tokens in C

The tokens of C language can be classified into six types based on the functions they are used to perform. The types of C tokens are as follows:



1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

# 1. C Token – Keywords

The keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. You cannot redefine keywords. However, you can specify the text to be substituted for keywords before compilation by using C preprocessor directives. **C** language supports **32** keywords which are given below:

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



## 2. C Token – Identifiers

Identifiers are used as the general terminology for the naming of variables, functions, and arrays. These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore(\_) as a first character. Identifier names must differ in spelling and case from any keywords. You cannot use keywords as identifiers; they are reserved for special use. Once declared, you can use the identifier in later program statements to refer to the associated value. A special identifier called a statement label can be used in goto statements.

### Rules for Naming Identifiers

Certain rules should be followed while naming c identifiers which are as follows:

- They must begin with a letter or underscore(\_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only the first 31 characters are significant.



### **3. C Token – Constants**

The constants refer to the variables with fixed values. They are like normal variables but with the difference that their values can not be modified in the program once they are defined.

Constants may belong to any of the data types.

In C programming language, identifiers are the building blocks of a program. Identifiers are unique names that are assigned to variables, structs, functions, and other entities. They are used to uniquely identify the entity within the program. In the below example “section” is an identifier assigned to the string type value.

A **variable in C language** is the name associated with some memory location to store data of different types. There are many types of variables in C depending on the scope, storage class, lifetime, type of data they store, etc. A variable is the basic building block of a C program that can be used in expressions as a substitute in place of the value it stores.



*Variable Syntax Breakdown*

There are 3 aspects of defining a variable:

1. Variable Declaration
2. Variable Definition
3. Variable Initialization

## 1. C Variable Declaration

Variable declaration in C tells the compiler about the existence of the variable with the given name and data type. When the variable is declared, an entry in symbol table is created and memory will be allocated at the time of initialization of the variable.

## 2. C Variable Definition

In the definition of a C variable, the compiler allocates some memory and some value to it. A defined variable will contain some random garbage value till it is not initialized.



## **Difference between Variable Initialization and Assignment**

Initialization occurs when a variable is first declared and assigned an initial value. This usually happens during the declaration of the variable. On the other hand, assignment involves setting or updating the value of an already declared variable, and this can happen multiple times after the initial initialization.

# 1. Local Variables in C

A **Local variable in C** is a variable that is declared inside a function or a block of code. Its scope is limited to the block or function in which it is declared.

## Example of Local Variable in C




```
1 // C program to declare and print local variable inside a
2 // function.
3 #include <stdio.h>
4
5 void function()
6 {
7     int x = 10; // local variable
8     printf("%d", x);
9 }
10
11 int main() { function(); }
```



## 2. Global Variables in C

A **Global variable in C** is a variable that is declared outside the function or a block of code. Its scope is the whole program i.e. we can access the global variable anywhere in the C program after it is declared.

### Example of Global Variable in C



```
1 // C program to demonstrate use of global variable
2 #include <stdio.h>
3
4 int x = 20; // global variable
5
6 void function1() { printf("Function 1: %d\n", x); }
7
8 void function2() { printf("Function 2: %d\n", x); }
9
10 int main()
11 {
12
13     function1();
14     function2();
15     return 0;
16 }
```

### 3. Static Variables in C

A static variable in C is a variable that is defined using the **static** keyword. It can be defined only once in a C program and its scope depends upon the region where it is declared (can be **global or local**).

The **default value** of static variables is **zero**.

## 4. Automatic Variable in C

All the **local** variables are **automatic** variables **by default**. They are also known as auto variables.

Their scope is **local** and their lifetime is till the end of the **block**. If we need, we can use the **auto** keyword to define the auto variables.

The default value of the auto variables is a garbage value.

## 5. External Variables in C

External variables in C can be shared between multiple C files. We can declare an external variable using the extern keyword.

## 6. Register Variables in C

**Register variables in C** are those variables that are stored in the **CPU register** instead of the conventional storage place like RAM. Their scope is **local** and exists till the **end** of the **block** or a function.

These variables are declared using the [register](#) keyword.

The default value of register variables is a **garbage value**.

| Constant  | Literals  |
|---|---|
| Constants are variables that cannot be modified once declared.                                  | Literals are the fixed values that define themselves.                           |
| Constants are defined by using the const keyword in C. They store literal values in themselves. | They themselves are the values that are assigned to the variables or constants. |
| We can determine the address of constants.  | We cannot determine the address of a literal except string literal.             |
| They are lvalues.   | They are rvalues.   |
| Example: <code>const int c = 20.</code>   | Example: 24,15.5, 'a', "Geeks", etc.  |




## Defining Constant using #define Preprocessor

We can also define a constant in C using [#define preprocessor](#). The constants defined using #define are macros that behave like a constant. These constants are not handled by the compiler, they are handled by the preprocessor and are replaced by their value before compilation.

```
#define const_name value
```

### Example of Constant Macro



```
1 // C Program to define a constant using #define
2 #include <stdio.h>
3 #define pi 3.14
4
5 int main()
6 {
7
8     printf("The value of pi: %.2f", pi);
9     return 0;
10 }
```

| Types                          | Description   | Data Types                     |
|--------------------------------|---|--------------------------------|
| Primitive Data Types           | Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc. | int, char, float, double, void |
| <u>Derived Types</u>           | The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.                           | array, pointers, function      |
| <u>User Defined Data Types</u> | The user-defined data types are defined by the user himself.  | structure, union, enum         |

## Implicit Type Conversion

An implicit type conversion is automatically performed by the compiler when differing data types are intermixed in an expression.

An implicit type conversion is performed without programmer's intervention.

Example:  
a, b = 5, 25.5  
c = a + b

## Explicit Type Conversion

An explicit type conversion is user-defined conversion that forces an expression to be of specific type.

An explicit type conversion is specified explicitly by the programmer.

Example:  
a, b = 5, 25.5  
c = int(a + b)

## What is a C Operator?

An operator in C can be defined as the symbol that helps us to perform some specific mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands. So we can say that the operators are the symbols that perform operations on operands.

## Operators in C

|                    | Operators             | Type                            |
|--------------------|-----------------------|---------------------------------|
| Unary Operator →   | ++, --                | Unary Operator                  |
| Binary Operator {  | +, -, *, /, %         | Arithmetic Operator             |
|                    | <, <=, >, >=, ==, !=  | Relational Operator             |
|                    | &&,   , !             | Logical Operator                |
|                    | &,  , <<, >>, ~, ^    | Bitwise Operator                |
|                    | =, +=, -=, *=, /=, %= | Assignment Operator             |
| Ternary Operator → | ?:                    | Ternary or Conditional Operator |

# Types of Operators in C

C language provides a wide range of functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

## sizeof Operator

- sizeof is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by size\_t.
- Basically, the sizeof the operator is used to compute the size of the variable or datatype.



## Comma Operator ( , )

- The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator.

## Conditional Operator ( ? : )

- The conditional operator is the only ternary operator in C++.
- Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
- We may replace the use of if..else statements with conditional operators.

## addressof (&) and Dereference (\*) Operators

- Pointer operator & returns the address of a variable. For example &a; will give the actual address of the variable.
- The pointer operator \* is a pointer to a variable. For example \*var; will pointer to a variable var.

The format specifier in C is used to tell the compiler about the type of data to be printed or scanned in input and output operations. They always start with a **%** symbol and are used in the formatted string in functions like `printf()`, `scanf`, `sprintf()`, etc.

| Data Type |                  | Format        |
|-----------|------------------|---------------|
| Integer   | Integer          | %d            |
|           | Short            | %d            |
|           | Short unsigned   | %u            |
|           | Long             | %ld           |
|           | Long assigned    | %lu           |
|           | Hexadecimal      | %x            |
|           | Long hexadecimal | %lx           |
|           | Octal            | %O (letter O) |
|           | long octal       | %lo           |
| Real      | float,double     | %f, %lf, %g   |
| Character |                  | %c            |
| String    |                  | %s            |

## Why &?

While scanning the input, scanf needs to store that input data somewhere. To store this input data, scanf needs to know the memory location of a variable. And here comes the ampersand to rescue.

- & is also called as address of the operator.
- For example, &var is the address of var.



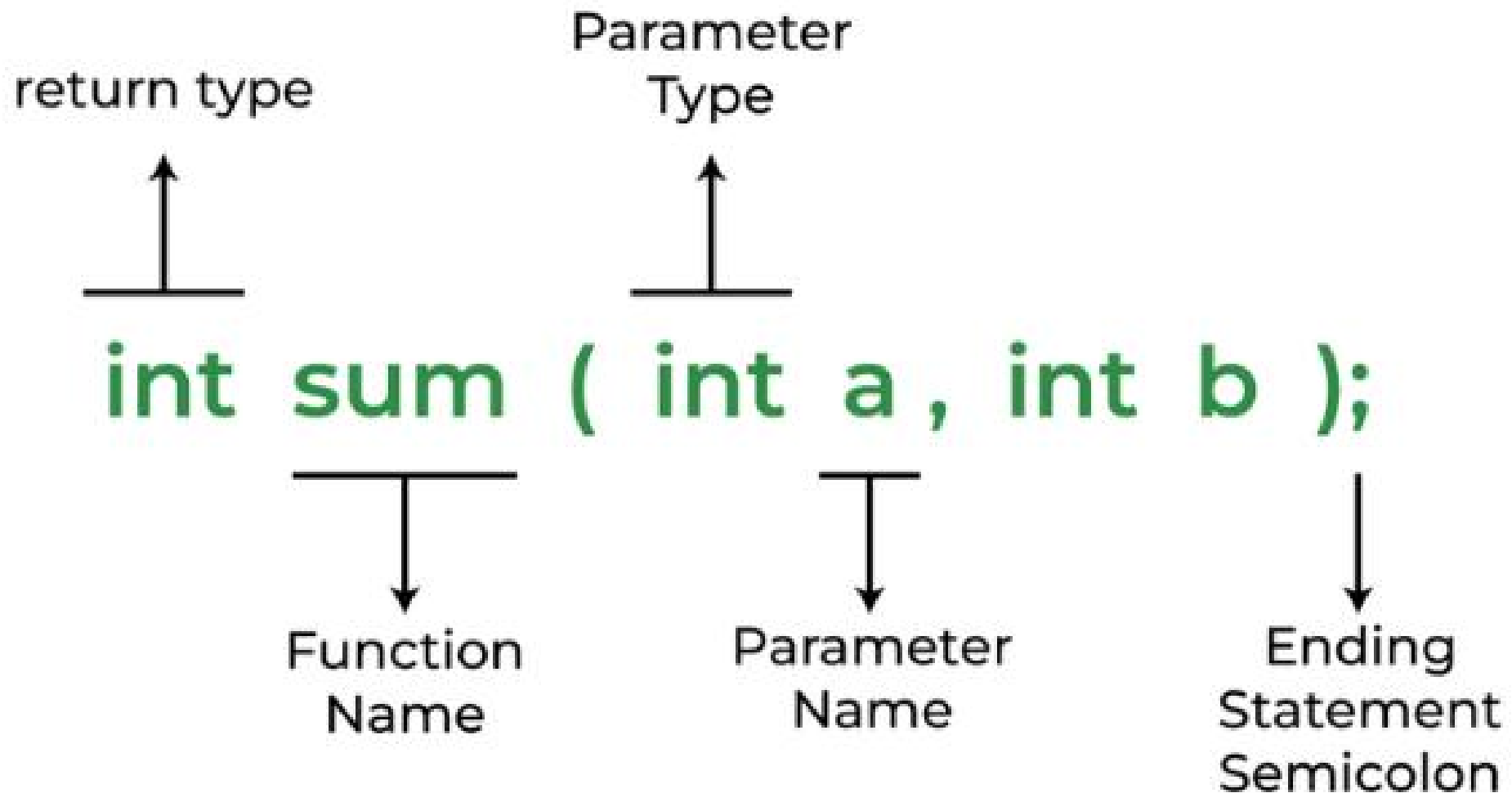
The escape sequence in C is the characters or the sequence of characters that can be used inside the string literal. The purpose of the escape sequence is to represent the characters that cannot be used normally using the keyboard. Some escape sequence characters are the part of ASCII charset but some are not.

Different escape sequences represent different characters but the output is dependent on the compiler you are using.

The **continue statement** in C is a jump statement used to skip the current iteration of a loop and continue with the next iteration. It is used inside loops (for, while, or do-while) along with the conditional statements to bypass the remaining statements in the current iteration and move on to next iteration.

The break in C is a loop control statement that breaks out of the loop when encountered. It can be used inside loops or switch statements to bring the control out of the block. The break statement can only break out of a single loop at a time.

A **function in C** is a set of statements that when called perform some specific tasks. It is the basic building block of a C program that provides modularity and code reusability. The programming statements of a function are enclosed within **{ }** **braces**, having certain meanings and performing certain operations. They are also called subroutines or procedures in other languages.



```
1
2 #include <stdio.h>
3 void func(); // function declaration
4
5 //function definition
6 void func2(){
7     printf("hello, ")
8 }
9
10 int main(){
11     func();
12     func2();
13     return 0;
14 }
15 //function definition
16 void func(){
17     printf("hello, ")
18 }
19
```

## Conditions of Return Types and Arguments

In C programming language, functions can be called either with or without arguments and might return values. They may or might not return values to the calling functions.

1. Function with no arguments and no return value
2. Function with no arguments and with return value
3. Function with argument and with no return value
4. Function with arguments and with return value



A library function is also referred to as a “**built-in function**”. A compiler package already exists that contains these functions, each of which has a specific meaning and is included in the package. Built-in functions have the advantage of being directly usable without being defined, whereas user-defined functions must be declared and defined before being used.

| Call By Value  | Call By Reference   |
|--|---|
| While calling a function, we pass the values of variables to it. Such functions are known as "Call By Values".   | While calling a function, instead of passing the values of variables, we pass the address of variables(location of variables) to the function known as "Call By References. |
| In this method, the value of each variable in the calling function is copied into corresponding dummy variables of the called function.                    | In this method, the address of actual variables in the calling function is copied into the dummy variables of the called function.  |
| With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function. | With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them.   |
| In call-by-values, we cannot alter the values of actual variables through function calls.  | In call by reference, we can alter the values of variables through function calls.  |
| Values of variables are passed by the Simple technique.  | Pointer variables are necessary to define to store the address values of variables.   |
| This method is preferred when we have to pass some small values that should not change.  | This method is preferred when we have to pass a large amount of data to the function.   |
| Call by value is considered safer as original data is preserved  | Call by reference is risky as it allows direct modification in original data  |

```
1 #include<stdio.h>
2 int func(int x){ // here x is formal parameter
3     return x;
4 }
5 int main(){
6     int a = 10;
7     printf("%d", func(a)); // here a is actual parameter
8     return 0;
9 }
```

```

1  #include<stdio.h>
2
3  int func(int *x){
4      *x +=10;
5      return *x;
6  } //call by reference
7
8  int main(){
9      int num = 9;
10     func(&num); // function calling
11     printf("%d", num);
12     return 0;
13 }

```

input

19

...Program finished with exit code 0  
Press ENTER to exit console.

```

1  #include<stdio.h>
2
3  int func(int x){
4      x +=10;
5      return x;
6  } //call by value
7
8  int main(){
9      int num = 9;
10     func(num);
11     printf("%d", num);
12     return 0;
13 }

```

input

9

...Program finished with exit code 0  
Press ENTER to exit console.

## Properties of Arrays in C

It is very important to understand the properties of the C array so that we can avoid bugs while using it. The following are the main [properties of an array in C](#):

### 1. Fixed Size

The array in C is a fixed-size collection of elements. The size of the array must be known at the compile time and it cannot be changed once it is declared.

### 2. Homogeneous Collection

We can only store one type of element in an array. There is no restriction on the number of elements but the type of all of these elements must be the same.

### 3. Indexing in Array

The array index always starts with 0 in C language. It means that the index of the first element of the array will be 0 and the last element will be  $N - 1$ .

### 4. Dimensions of an Array

A dimension of an array is the number of indexes required to refer to an element in the array. It is the number of directions in which you can grow the array size.

### 5. Contiguous Storage

All the elements in the array are stored continuously one after another in the memory. It is one of the defining properties of the array in C which is also the reason why random access is possible in the array.

### 6. Random Access

The array in C provides random access to its element i.e we can get to a random element at any index of the array in constant time complexity just by using its index number.

### 7. No Index Out of Bounds Checking

There is no index out-of-bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.