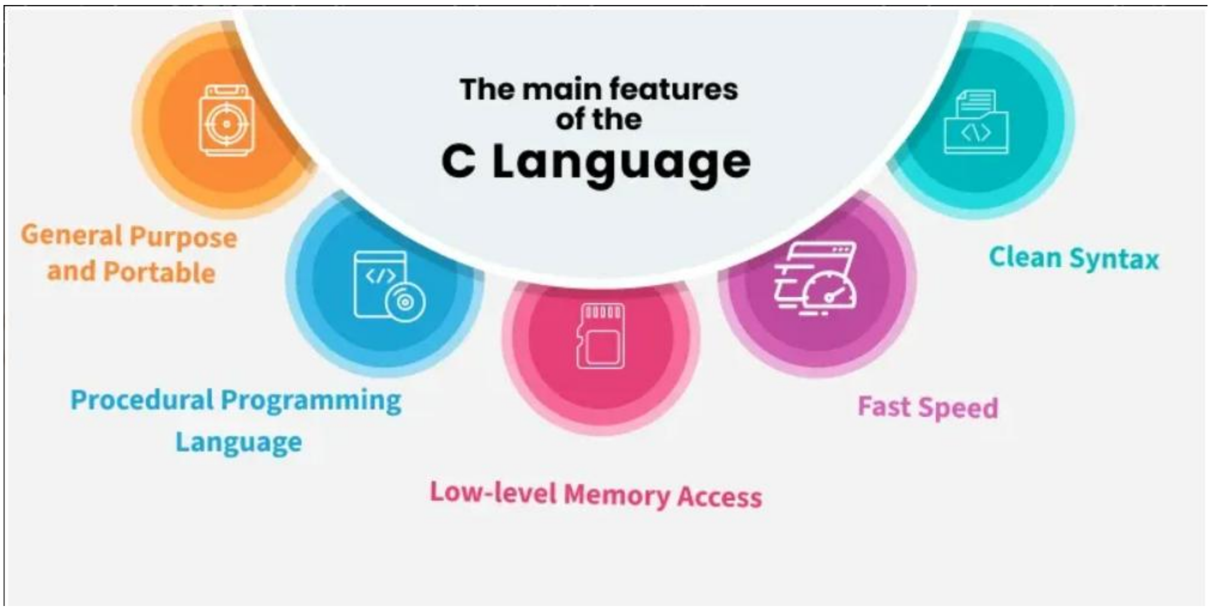
C Language Introduction

Last Updated: 23 Jul, 2025

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



Main features of C

Why Learn C?

- C is considered **mother of all programming languages** as many later languages like Java, PHP and JavaScript have borrowed syntax/features directly or indirectly from the C.
- If a person learns C programming first, it helps to learn any modern programming language as it provide a deeper **understanding of the fundamentals of programming** and underlying architecture of the operating system like pointers, working with memory locations etc.
- C is widely used in **operating systems**, embedded systems, compilers, databases, networking, game engines, and real-time systems for its efficiency to work in low resource environment and hardware-level support.

Writing First Program in C Language

This simple program demonstrates the basic structure of a C program. It will also help us understand the basic syntax of a C program.

```
#include <stdio.h>

int main(void)

{

   // This prints "Hello World"
```

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

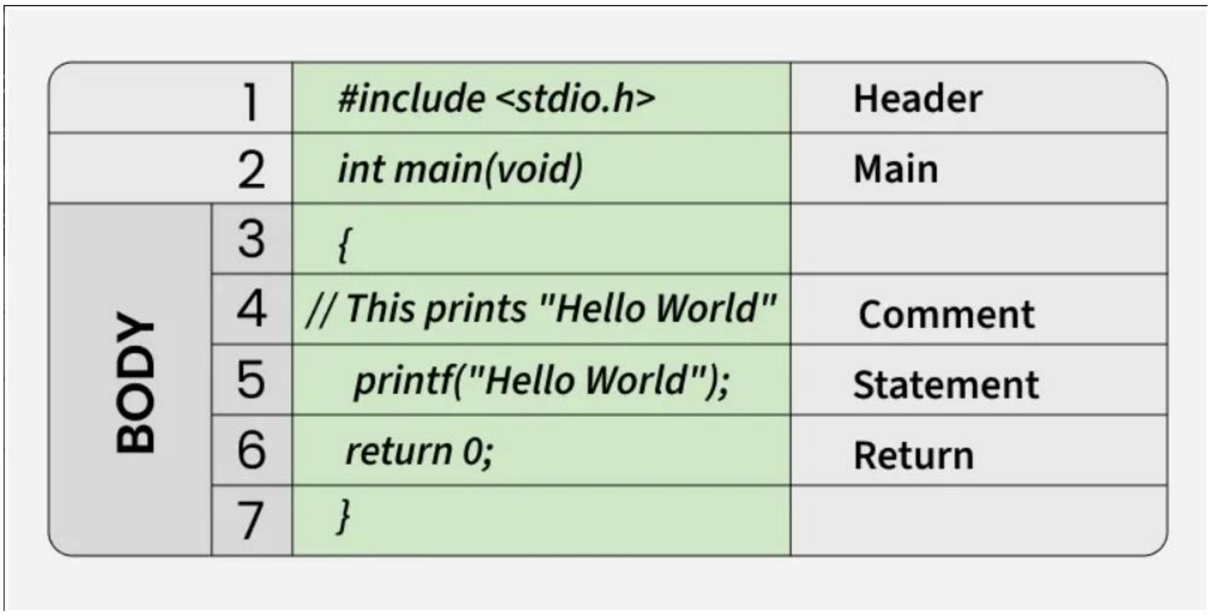
Output

```
Hello World
```

Let us analyse the structure of our program line by line.

Structure of the C program

After the above discussion, we can formally assess the basic structure of a C program. By structure, it is meant that any program can be written in this structure only. Writing a C program in any other structure will lead to a Compilation Error. The structure of a C program is as follows:



	1	#include <stdio.h>	Header
	2	int main(void)	Main
BODY	3	{	
	4	// This prints "Hello World"	Comment
	5	printf("Hello World");	Statement
	6	return 0;	Return
	7	}	

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

The first component is 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 preprocesses code of stdio.h to our file. The .h files are called header files in C.
Some of the C Header files:

- stddef.h - Defines several useful types and macros.
- stdint.h - Defines exact width integer types.
- stdio.h - Defines core input and output functions
- stdlib.h - Defines numeric conversion functions, pseudo-random number generator, and memory allocation
- string.h - Defines string handling functions

- math.h - Defines common mathematical functions.

Main Method Declaration - Line 2 [int main()]

The next part of a C program is 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.

Body of Main Method - Line 3 to Line 6 [enclosed in {}]

The body of the main method in the C program refers to statements that are a part of the main function. It can be anything like manipulations, searching, sorting, printing, etc. A pair of curly brackets define the body of a function. All functions must start and end with curly brackets.

Comment - Line 7[// This prints "Hello World"]

The comments are used for the documentation of the code or to add notes in your program that are ignored by the compiler and are not the part of executable program .

Statement - Line 4 [printf("Hello World");]

Statements are the instructions given to the compiler. In C, a statement is always terminated by a **semicolon (;).** In this particular case, we use printf() function to instruct the compiler to display "Hello World" text on the screen.

Return Statement - Line 5 [return 0;]

The last part of any C function is the return statement. The return statement refers to the return values from a function. This return statement and return value depend upon the return type of the function. The return statement in our program returns the value from main(). The returned value may be used by an operating system to know the termination status of your program. The value 0 typically means successful termination.

Execute C Programs

In order to execute the above program, we need to first compile it using a compiler and then we can run the generated executable. There are online IDEs available for free like GeeksforGeeksIDE, that can be used to start development in C without installing a compiler.

If you want to run the programs in your computer, then you will have to create a development environment for C. A development environment consists mainly consists of a text editor, which is used to write the code, and a compiler that converts the written code into the executable file. Refer to this article - Setting Up C Development Environment

Difference Between C and C++

C++ was created to add the OOPs concept into the C language so they both have very similar syntax with a few differences. The following are some of the main differences between C and C++ Programming languages.

Feature	C	C++
Paradigm	Procedural programming	Procedural, Object-Oriented, Generic
OOPs	Not Supported	Supports OOPs concepts with classes, inheritance, polymorphism, and encapsulation.
Memory Management	Manual using malloc(), calloc(), free() etc.	Both Manual and Automatic (using new, delete for dynamic allocation and deallocation)
Function Overloading	Not Supported (function names must be unique.)	Supports function overloading.
Operator Overloading	Not Supported (every operator performs unique operation.)	Supports operator overloading, allows custom behaviors for operators like +, -, *, etc.
Access Control	Doesn't have any access control mechanism.	Supports access control using keywords like private, public , protected.

Application of C Language

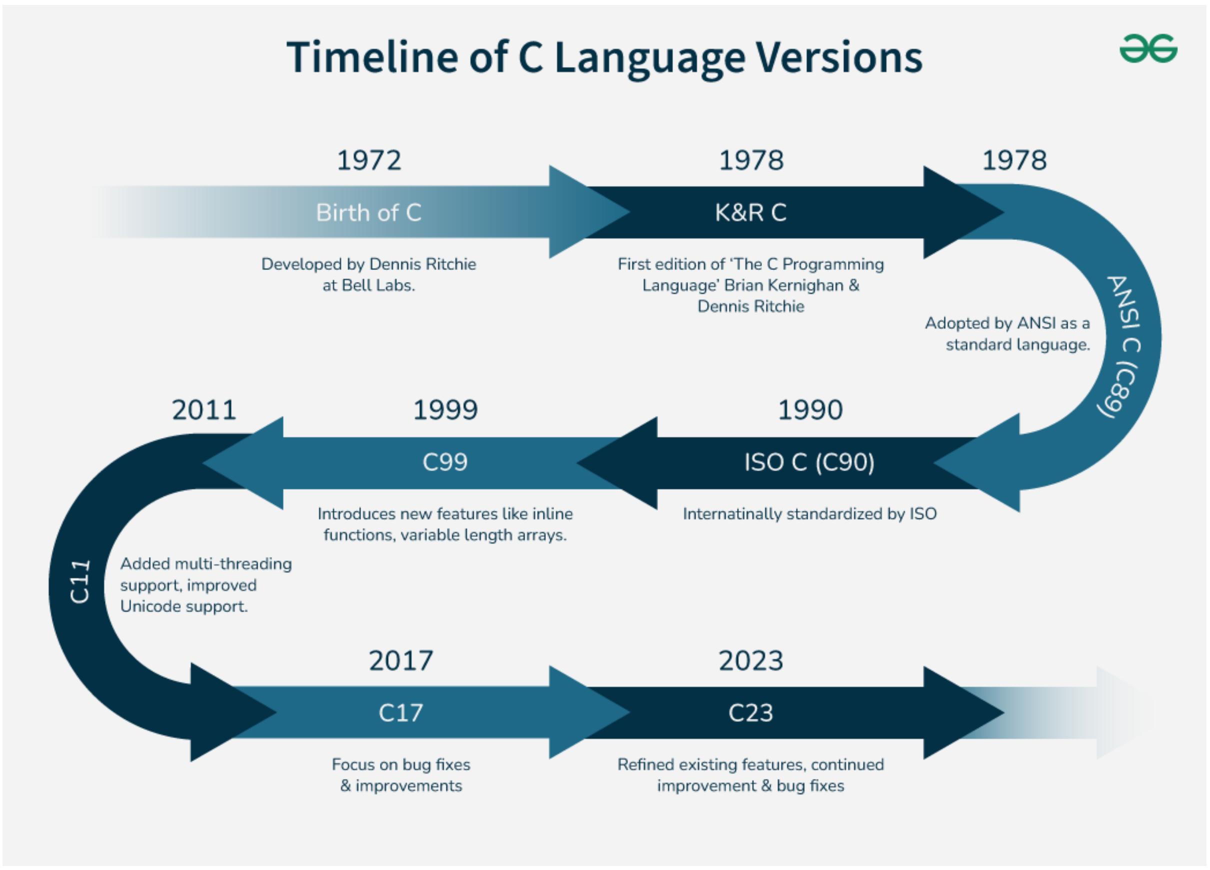
C language is being used since the early days of computer programming and still is used in wide variety of applications such as:

- C is used to develop core components of **operating systems** such as Windows, Linux, and macOS.
- C is applied to program **embedded systems** in small devices such as washing machines, microwave ovens, and printers.
- C is utilized to create efficient and quick **game engines**. For example, the Doom video game engine was implemented using C.
- C is employed to construct **compilers**, **assemblers**, and **interpreters**. For example, the CPython interpreter is written partially using C.
- C is applied to develop efficient **database engines**. The MySQL database software is implemented using C and C++.

- C is employed to create **programs** for devices and sensors of **Internet of Things (IoT)**. A common example is a house automation system comprising temperature sensors and controllers that is often prepared with C.
- C is employed for creating **lightweight and speedy desktop applications**. The widely used text editor Notepad++ employs C for performance-sensitive sections.

History Of C

C is a **general-purpose** procedural programming language developed by Dennis Ritchie in 1972 at Bell Labs to build the **UNIX** operating system. It provides **low-level memory** access, high performance, and portability, making it ideal for system programming. Over the years, C has evolved through standards like **ANSI C**, C99, C11, and C23, adding modern features while retaining its simplicity. Read more about history here.


Timeline of C Language Versions
1972
Birth of C
Developed by Dennis Ritchie at Bell Labs.
1978
K&R C
First edition of 'The C Programming Language' Brian Kernighan & Dennis Ritchie
1978
ANSI C (C89)
Adopted by ANSI as a standard language.
1990
ISO C (C90)
Internatinally standardized by ISO
1999
C99
Introduces new features like inline functions, variable length arrays.
2011
C11
Added multi-threading support, improved Unicode support.
2017
C17
Focus on bug fixes & improvements
2023
C23
Refined existing features, continued improvement & bug fixes

Features of C Programming Language

Last Updated: 14 Jul, 2025

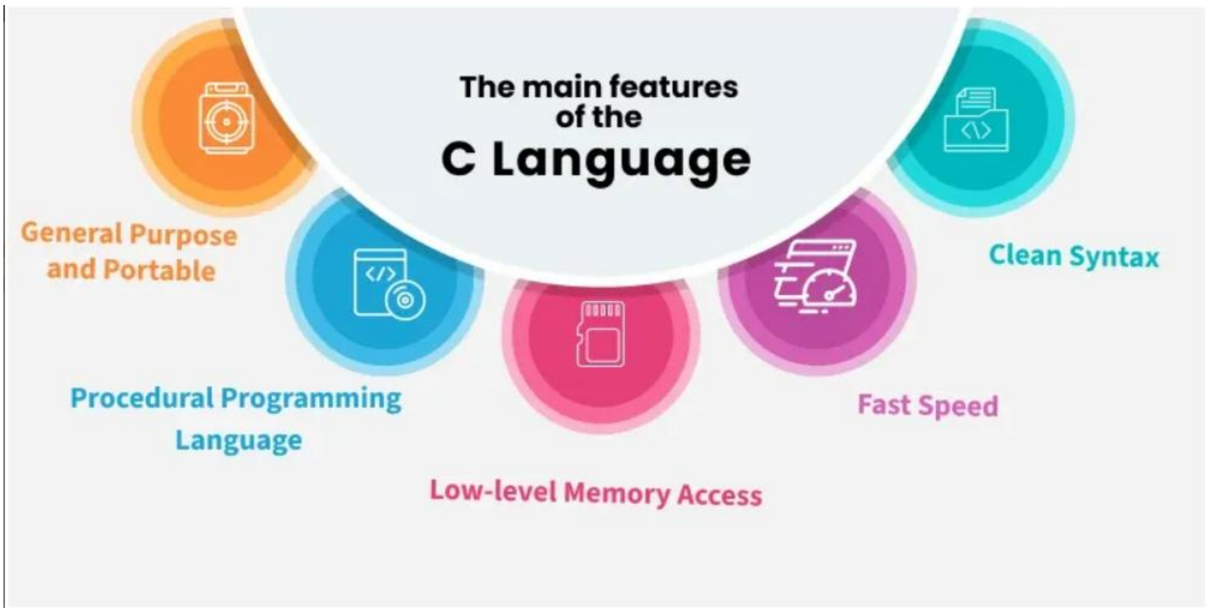
C is a procedural programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system.

The main features of C language include low-level access to memory, a simple set of keywords, and a clean style, these features make C language suitable for system programming like an operating system or compiler development.

What are the Most Important Features of C Language?

Here are some of the most important features of the C language:

1. Procedural Language
2. Fast and Efficient
3. Modularity
4. Statically Type
5. General-Purpose Language
6. Rich set of built-in Operators
7. Libraries with Rich Functions
8. Middle-Level Language
9. Portability
10. Easy to Extend



Let discuss these features one by one:

1. Procedural Language

In a procedural language like C step by step, predefined instructions are carried out. C program may contain more than one function to perform a particular task. New people to programming will think that this is the only way a particular programming language works. There are other programming paradigms as well in the programming world. Most of the commonly used paradigm is an object-oriented programming language.

2. Fast and Efficient

Newer languages like Java, python offer more features than c programming language but due to additional processing in these languages, their performance rate gets down effectively. C programming language as the middle-level language provides programmers access to direct manipulation with the computer hardware but higher-level languages do not allow this. That’s one of the reasons C language is considered the first choice to start learning programming languages. It’s fast because statically typed languages are faster than dynamically typed languages.

3. Modularity

The concept of storing C programming language code in the form of libraries for further future uses is known as modularity. This programming language can do very little on its own most of its power is held by its libraries. C language has its own library to solve common problems.

4. Statically Type

C programming language is a statically typed language. Meaning the type of variable is checked at the time of compilation but not at run time. This means each time a programmer types a program they have to mention the type of variables used.

5. General-Purpose Language

From system programming to photo editing software, the C programming language is used in various applications. Some of the common applications where it’s used are as follows:

- Operating systems: Windows, Linux, iOS, Android, macOS
- Databases: PostgreSQL, Oracle, MySQL, MS SQL Server, etc.

6. Rich set of built-in Operators

It is a diversified language with a rich set of built-in operators which are used in writing complex or simplified C programs.

7. Libraries with Rich Functions

Robust libraries and functions in C help even a beginner coder to code with ease.

8. Middle-Level Language

As it is a middle-level language so it has the combined form of both capabilities of assembly language and features of the high-level language.

9. Portability

C language is lavishly portable as programs that are written in C language can run and compile on any system with either no or small changes.

10. Easy to Extend

Programs written in C language can be extended means when a program is already written in it then some more features and operations can be added to it.

C Programming Language Standard

Last Updated: 23 Jul, 2025

Introduction:

The C programming language has several standard versions, with the most commonly used ones being C89/C90, C99, C11, and C18.

1. C89/C90 (ANSI C or ISO C) was the first standardized version of the language, released in 1989 and 1990, respectively. This standard introduced many of the features that are still used in modern C programming, including data types, control structures, and the standard library.
2. C99 (ISO/IEC 9899:1999) introduced several new features, including variable-length arrays, flexible array members, complex numbers, inline functions, and designated initializers. This standard also includes several new library functions and updates to existing ones.
3. C11 (ISO/IEC 9899:2011) introduced several new features, including _Generic, static_assert, and the atomic type qualifier. This standard also includes several updates to the library, including new functions for math, threads, and memory manipulation.
4. C18 (ISO/IEC 9899:2018) includes updates and clarifications to the language specification and the library.
5. C23 standard (ISO/IEC 9899:2023) is the latest revision and include better support for the const qualifier, new and updated library functions, and further optimizations for compiler implementations.

When writing C code, it's important to know which standard version is being used and to write code that is compatible with that standard. Many compilers support multiple standard versions, and it's often possible to specify which version to use with a compiler flag or directive.

Here are some advantages and disadvantages of using the C programming language:

Advantages:

1. Efficiency: C is a fast and efficient language that can be used to create high-performance applications.
2. Portability: C programs can be compiled and run on a wide range of platforms and operating systems.
3. Low-level access: C provides low-level access to system resources, making it ideal for systems programming and developing operating systems.
4. Large user community: C has a large and active user community, which means there are many resources and libraries available for developers.
5. Widely used: C is a widely used language, and many modern programming languages are built on top of it.

Disadvantages:

1. Steep learning curve: C can be difficult to learn, especially for beginners, due to its complex syntax and low-level access to system resources.
2. Lack of memory management: C does not provide automatic memory management, which can lead to memory leaks and other memory-related bugs if not handled properly.

3. No built-in support for object-oriented programming: C does not provide built-in support for object-oriented programming, making it more difficult to write object-oriented code compared to languages like Java or Python.

4. No built-in support for concurrency: C does not provide built-in support for concurrency, making it more difficult to write multithreaded applications compared to languages like Java or Go.

5. Security vulnerabilities: C programs are prone to security vulnerabilities, such as buffer overflows, if not written carefully.
   Overall, C is a powerful language with many advantages, but it also requires a high degree of expertise to use effectively and has some potential drawbacks, especially for beginners or developers working on complex projects.

Importance:

important for several reasons:

1. Choosing the right programming language: Knowing the advantages and disadvantages of C can help developers choose the right programming language for their projects. For example, if high performance is a priority, C may be a good choice, but if ease of use or built-in memory management is important, another language may be a better fit.

2. Writing efficient code: Understanding the efficiency advantages of C can help developers write more efficient and optimized code, which is especially important for systems programming and other performance-critical applications.

3. Avoiding common pitfalls: Understanding the potential disadvantages of C, such as memory management issues or security vulnerabilities, can help developers avoid common pitfalls and write safer, more secure code.

4. Collaboration and communication: Knowing the advantages and disadvantages of C can also help developers communicate and collaborate effectively with others on their team or in the wider programming community.

The idea of this article is to introduce C standard.

What to do when a C program produces different results in two different compilers?
For example, consider the following simple C program.

```
void main() {  }
```

The above program fails in GCC as the return type of main is void, but it compiles in Turbo C. How do we decide whether it is a legitimate C program or not?

Consider the following program as another example. It produces different results in different compilers.

```
// C++ Program to illustrate the difference in different

// compiler execution

#include <stdio.h>
```

```
int main()

{

   int i = 1;

   printf("%d %d %d\n", i++, i++, i);

   return 0;

}
```

```
2 1 3 - using g++ 4.2.1 on Linux.i686
1 2 3 - using SunStudio C++ 5.9 on Linux.i686
2 1 3 - using g++ 4.2.1 on SunOS.x86pc
1 2 3 - using SunStudio C++ 5.9 on SunOS.x86pc
1 2 3 - using g++ 4.2.1 on SunOS.sun4u
1 2 3 - using SunStudio C++ 5.9 on SunOS.sun4u
```

Source: Stackoverflow
Which compiler is right?
The answer to all such questions is C standard. In all such cases, we need to see what C standard says about such programs.

What is C standard?
The latest C standard is ISO/IEC 9899:2018, also known as C17 as the final draft was published in 2018. Before C11, there was C99. The C11 final draft is available here. See this for a complete history of C standards.

Can we know the **behavior of all programs from C standard?**
C standard leaves some behavior of many C constructs as undefined and some as unspecified to simplify the specification and allow some flexibility in implementation. For example, in C the use of any automatic variable before it has been initialized yields undefined behavior and order of evaluations of subexpressions is unspecified. This specifically frees the compiler to do whatever is easiest or most efficient, should such a program be submitted.

So what is the conclusion about above two examples?
Let us consider the first example which is "void main() {}", the standard says following about prototype of main().

> The function called at program startup is named main. The implementation declares no prototype for this function. It shall be defined with a return type of int and with no parameters:
>
>   int main(void) { /* ... */ }
>
> or with two parameters (referred to here as argc and argv, though any names may be used, as they are local to the function in which they are declared):
>
>   int main(int argc, char *argv[]) { /* ... */ }
>
> or equivalent;10) or in some other implementation-defined manner.

So the return type void doesn't follow the standard, and it's something allowed by certain compilers.

Let us talk about the second example. Note the following statement in C standard is listed under unspecified behavior.

> The order in which the function designator, arguments, and
> subexpressions within the arguments are evaluated in a function
> call (6.5.2.2).

What to do with programs whose behavior is undefined or unspecified in standard?
As a programmer, it is never a good idea to use programming constructs whose behavior is undefined or unspecified, such programs should always be discouraged. The output of such programs may change with the compiler and/or machine.

C Hello World Program

Last Updated : 12 Jul, 2025

The "Hello World" program is the first step towards learning any programming language. It is also one of the simplest programs that is used to introduce aspiring programmers to the programming language. It typically outputs the text "Hello, World!" to the console screen.

C Program to Print "Hello World"

To print the "Hello World", we can use the printf function from the **stdio.h** library that prints the given string on the screen. Provide the string "Hello World" to this function as shown in the below code:

```
// Header file for input output functions
#include <stdio.h>

// Main function: entry point for execution
int main() {

   // Writing print statement to print hello world
   printf("Hello World");

   return 0;
}
```

Output

```
Hello World
```

Explanation:

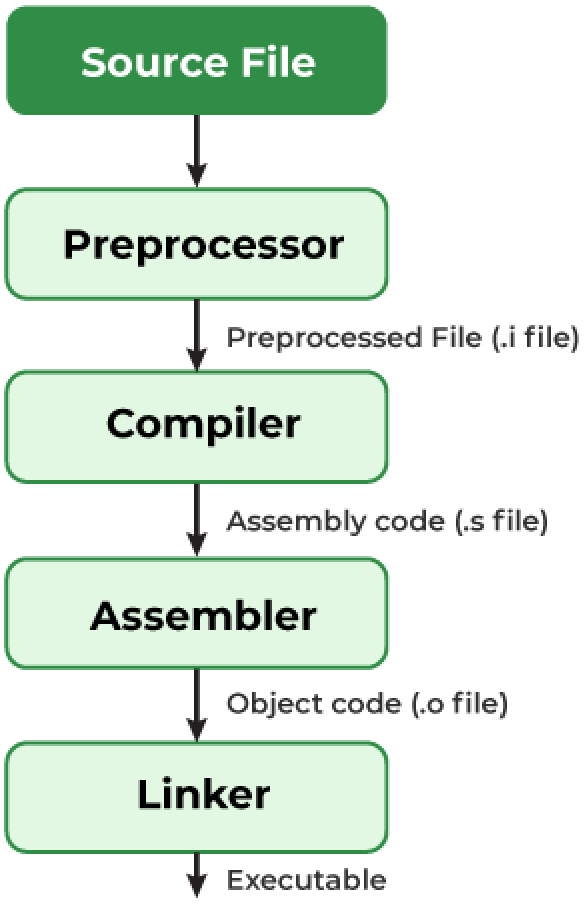
- **#include <stdio.h>** – This line includes the standard input-output library in the program.
- **int main()** – The main function where the execution of the program begins.
- **printf("Hello, World!\n");** – This function call prints "Hello, World!" followed by a new line.
- **return 0;** -This statement indicates that the program ended successfully.

Compiling a C Program: Behind the Scenes

Last Updated : 23 Jul, 2025

The compilation is the process of converting the source code of the C language into machine code. As C is a mid-level language, it needs a compiler to convert it into an executable code so that the program can be run on our machine.

The C program goes through the following phases during compilation:



Compilation Process in C

Understanding the compilation process in C helps developers optimize their programs.

How do we compile and run a C program?

We first need a compiler and a code editor to compile and run a C Program. The below example is of an Ubuntu machine with GCC compiler.

Step 1: Creating a C Source File

We first create a C program using an editor and save the file as filename.c In linux, we can use vi to create a file from the terminal using the command:

```
vi filename.c
```

In windows, we can use the Notepad to do the same. Then write a simple hello world program and save it.

Step 2: Compiling using GCC compiler

We use the following command in the terminal for compiling our **filename.c** source file.

```
gcc filename.c –o filename
```

We can pass many instructions to the GCC compiler to different tasks such as:

- The option -Wall enables all compiler's warning messages. This option is recommended to generate better code.
- The option -o is used to specify the output file name. If we do not use this option, then an output file with the name a.out is generated.

If there are no errors in our C program, the executable file of the C program will be generated.

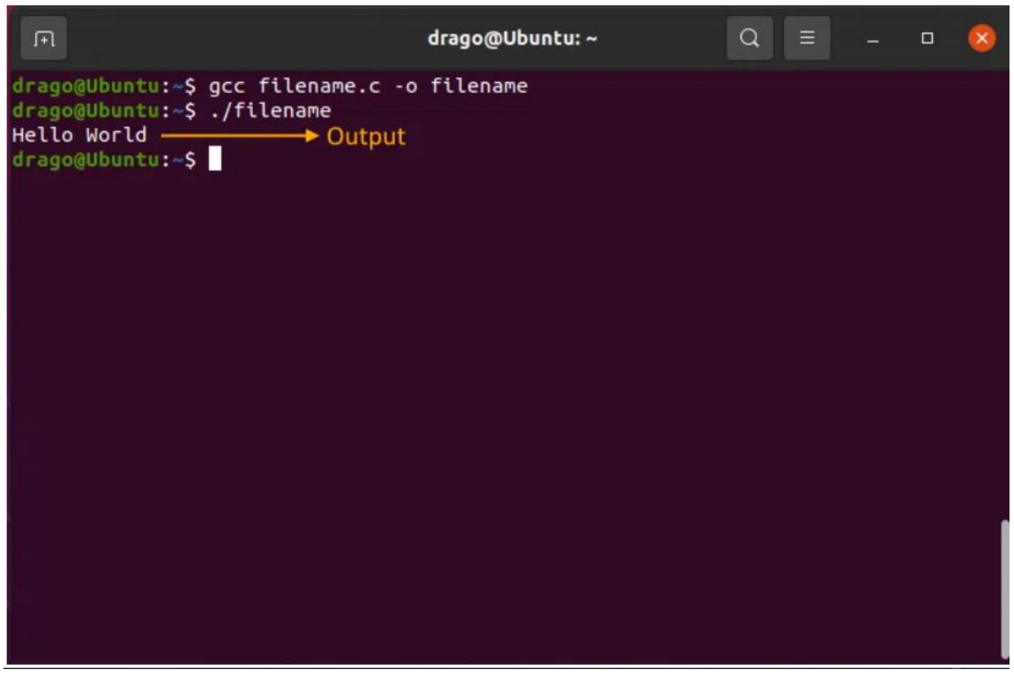
Step 3: Executing the program

After compilation executable is generated and we run the generated executable using the below command.

```
./filename // for linux

filename // for windows
```

The program will be executed, and the output will be shown in the terminal.



```
drago@Ubuntu:~$ gcc filename.c -o filename
drago@Ubuntu:~$ ./filename
Hello World
drago@Ubuntu:~$
```

What goes inside the compilation process?

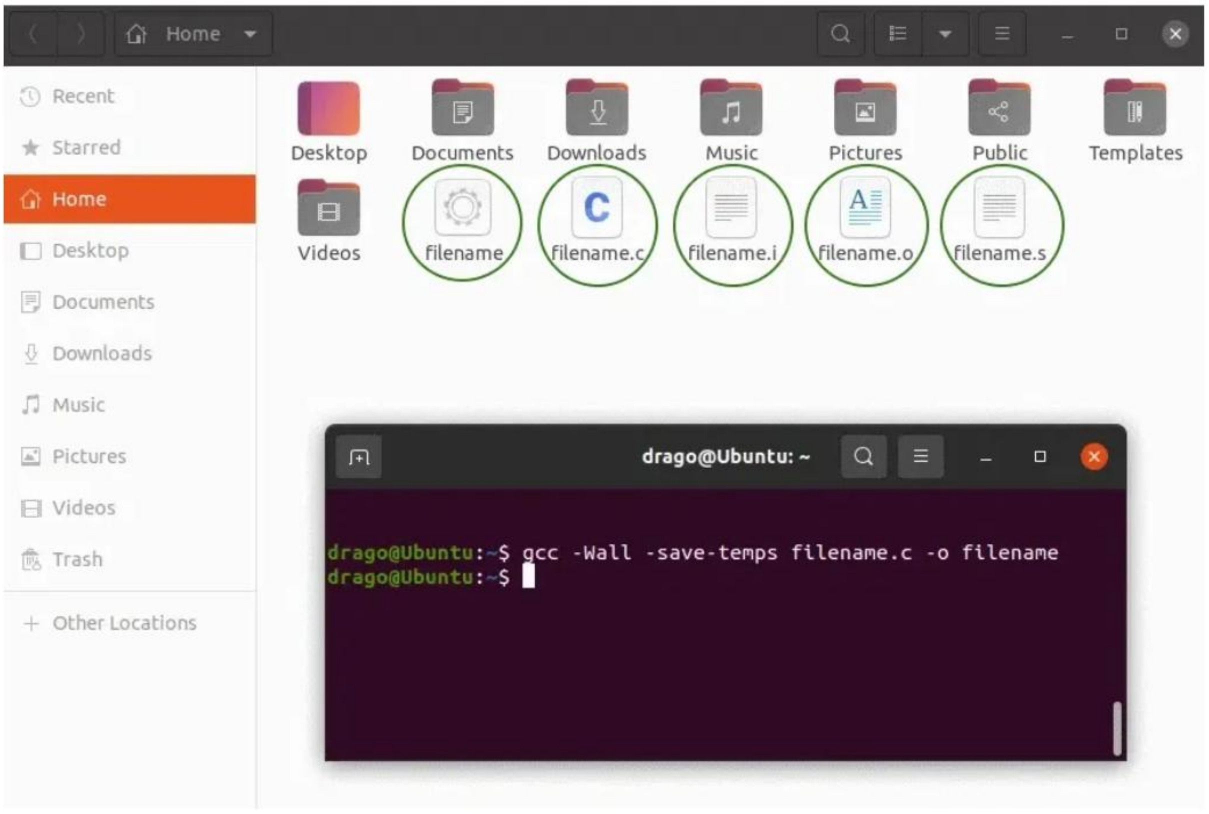
A compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. **Pre-processing**
2. **Compilation**
3. **Assembly**
4. **Linking**

By executing the below command while compiling the code, we get all intermediate files in the current directory along with the executable.

```
gcc -Wall -save-temps filename.c –o filename
```

The following screenshot shows all generated intermediate files.



```
drago@Ubuntu:~$ gcc -Wall -save-temps filename.c -o filename
drago@Ubuntu:~$
```

Let us one by one see what these intermediate files contain.

1. Pre-processing

This is the first phase through which source code is passed. This phase includes:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.
- Conditional compilation

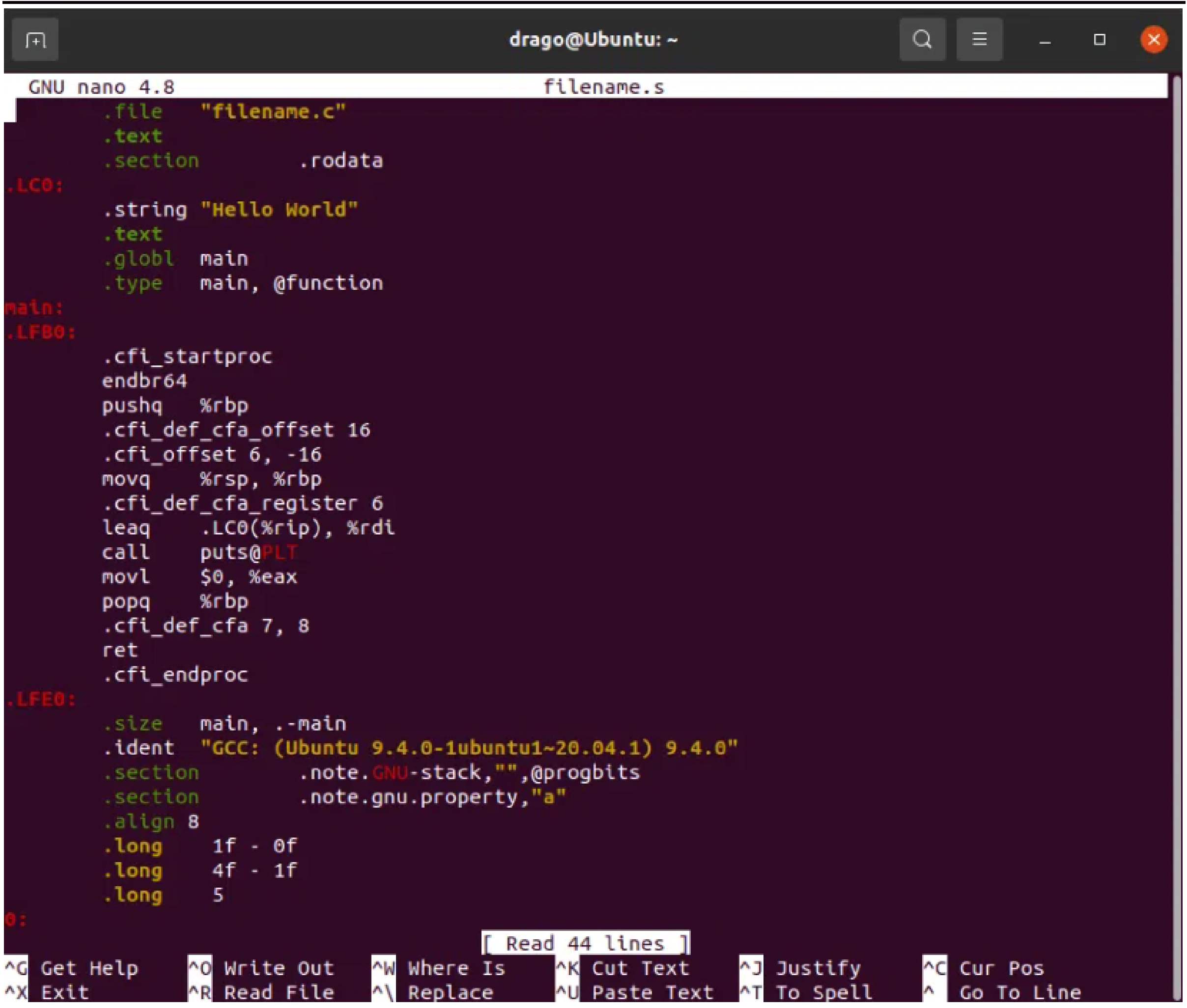
The pre-processed output is stored in the **filename.i**. Let's see what's inside filename.i: using **$vi filename.i**

In the above output, the source file is filled with lots and lots of info, but in the end, our code is preserved.

- printf contains now a + b rather than add(a, b) that's because macros have expanded.
- Comments are stripped off.
- **#include<stdio.h>** is missing instead we see lots of code. So header files have been expanded and included in our source file.

2. Compiling

The next step is to compile filename.i and produce an; intermediate compiled output file **filename.s**. This file is in assembly-level instructions. Let's see through this file using **$nano filename.s** terminal command.



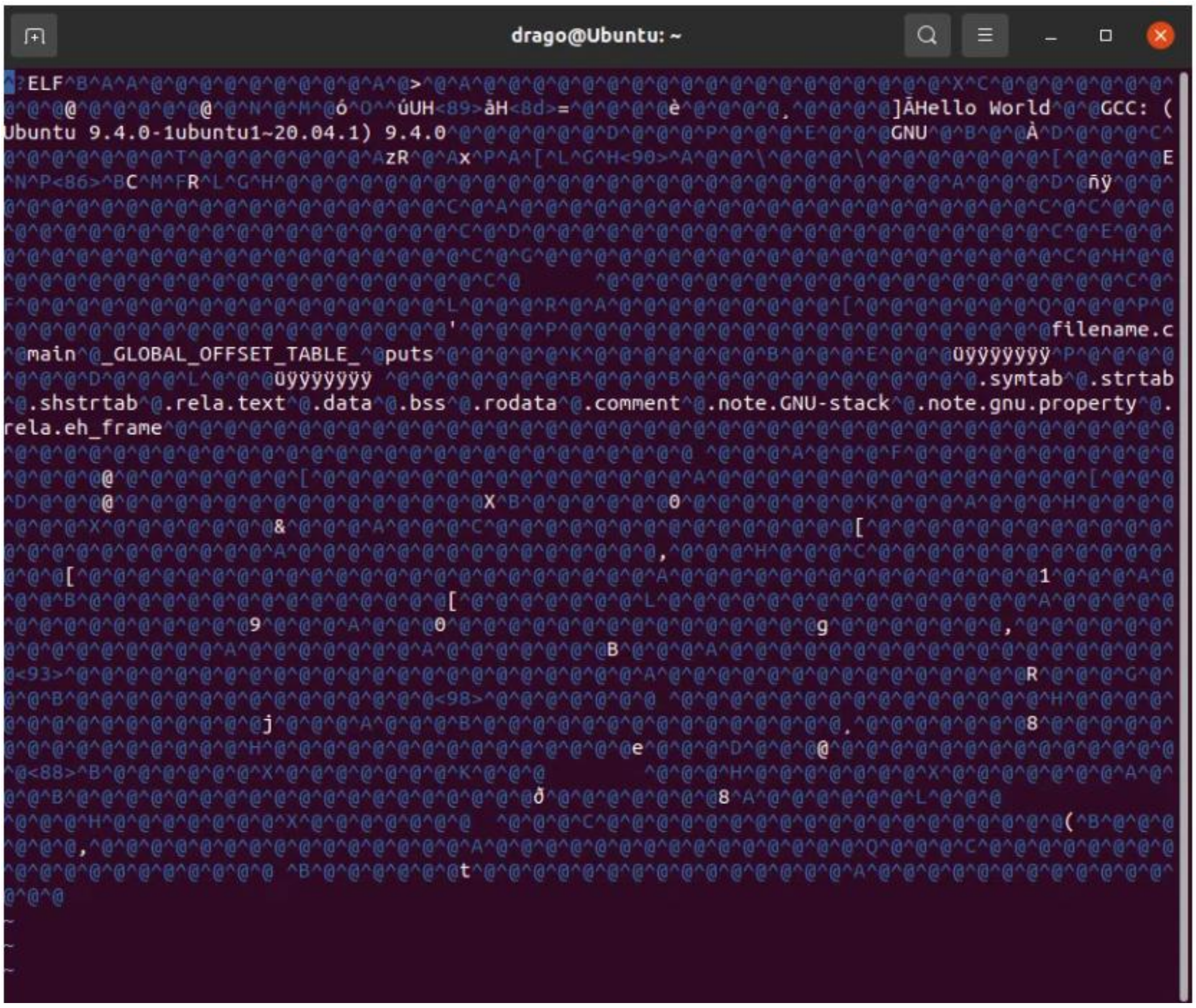
```
GNU nano 4.8                          filename.s
        .file   "filename.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
        .long    5
0:
                              [ Read 44 lines ]
^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell    ^_ Go To Line
```

Assembly Code File

The snapshot shows that it is in assembly language, which the assembler can understand.

3. Assembling

In this phase the filename.s is taken as input and turned into **filename.o** by the assembler. This file contains machine-level instructions. At this phase, only existing code is converted into machine language, and the function calls like printf() are not resolved. Let's view this file using **$vi filename.o**



```
^?ELF^B^A^A^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^X^C^@^@^@^@^@^@^
@^@^@@^@^@^@^@^@@^@^N^@^M^@ó^O^^úUH<89>åH<8d>=^@^@^@^@è^@^@^@^@¸^@^@^@^@]ÃHello World^@^@GCC: (
Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0^@^@^@^@^@^@^D^@^@^@^P^@^@^@^E^@^@^@GNU^@^B^@^@À^D^@^@^@^C^
@^@^@^@^@^@^@^T^@^@^@^@^@^@^@^AzR^@^Ax^P^A^[^L^G^H<90>^A^@^@^\^@^@^@^\^@^@^@^@^@^@^@^[^@^@^@^@E
^N^P<86>^BC^M^FR^L^G^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@ñÿ^@^@^
...
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@filename.c
^@main^@_GLOBAL_OFFSET_TABLE_^@puts^@^@^@^@^@^K^@^@^@^@^@^@^B^@^@^@^E^@^@^@üÿÿÿÿÿÿÿ^P^@^@^@^@
^@^@^@^D^@^@^@^L^@^@^@üÿÿÿÿÿÿÿ ^@^@^@^@^@^@^@^B^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@^@^@.symtab^@.strtab
^@.shstrtab^@.rela.text^@.data^@.bss^@.rodata^@.comment^@.note.GNU-stack^@.note.gnu.property^@.
rela.eh_frame^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
...
~
~
~
```

Binary Code

4. Linking

This is the final phase in which all the linking of function calls with their definitions is done. Linker knows where all these functions are implemented. Linker does some extra work also: it adds some extra code to our program which is required when the program starts and ends. For example, there is a code that is required for setting up the environment like passing command line arguments. This task can be easily verified by using **size filename.o** and **size filename**. Through these commands, we know how the output file increases from an object file to an executable file. This is because of the extra code that Linker adds to our program.

Linking can be of two types:

- **Static Linking**: All the code is copied to the single file and then executable file is created.
- **Dynamic Linking**: Only the names of the shared libraries is added to the code and then it is referred during the execution.

GCC by default does dynamic linking, so printf() is dynamically linked in above program.

C Comments

Last Updated: 11 Jul, 2025

The **comments in C** are human-readable notes in the source code of a C program used to make the program easier to read and understand. They are not a part of the executable program by the compiler or an interpreter.

Example:

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


   // This is a comment, the below
   // statement will not be executed
   // printf("Hi from comment");
   printf("Hello");
   return 0;
}
```

Output

```
Hello
```

Types of Comments in C

In C, there are two types of comments in C language:

- **Single-line Comments**
- **Multi-line Comments**

Single-line Comments

Single-line comments are used to comment out a single line of code or a part of it. The single line comments in C start with **two forward slashes (//)**, and everything after the slashes on that line is considered a comment. They are also called C++ Style comments as they were first introduced in C++ and later adopted in C also.

Syntax

```
// This is a single line comment
```

Example:

```
#include <stdio.h>


int main() {
    // This is a single-line comment explaining the variable x
    int x = 5;


    // Output the value of x
    printf("Value of x: %d\n", x);
    return 0;
}
```

Output

```
Value of x: 5
```

In this C program, the comments provide explanations about the code. The compiler ignores the comments and does not execute them.

We can also create a comment that displays at the end of a line of code using a single-line comment. But generally, it's better to practice putting the comment before the line of code.

```
#include <stdio.h>


int main() {
    // single line comment here


    printf("Hi"); // After line comment here
    return 0;
}
```

Output

```
Hi
```

Multi-line Comments

Multi-line comments in C are used write comments that span **more than one line.** They are generally used for longer descriptions or for commenting out multiple lines of code. In C language, these comments **begin with /*** and **end with */.** Everything between these markers is treated as a comment.

Syntax:

```
/* This is a multi-line comment
   which can span multiple lines */
```

Example:

```
#include <stdio.h>


int main() {


    /*
   This comment contains some code which
   will not be executed.
   printf("Code enclosed in Comment");
   */
   printf("Welcome to GeeksforGeeks");
   return 0;
}
```

Output

```
Welcome to GeeksforGeeks
```

Nesting Comments in C

In C language, comments cannot be nested. That means you cannot place a multi-line comment inside another multi-line comment in C language. If you try to do so, the compiler will treat the closing */ of the inner comment as the end of the entire multi-line comment. and the rest of the part after the **inner closing (*/)** will not be commented.

```
#include <stdio.h>


int main() {
   /* This is the start of a multi-line comment
      /* This is an inner multi-line comment */
      This line will cause an error because the compiler
      considers the above '*/' as the end of the comment block.
   */
   printf("This program will not compile.\n");
   return 0;
}
```

Output

```
./Solution.c: In function 'main':
./Solution.c:6:8: error: unknown type name 'This'
     This line will cause an error because the compiler
     ^
./Solution.c:6:18: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'will'
     This line will cause an error because the compiler
           ^
```

If nested comments are needed, it’s best to **use single-line comments** or comment out individual parts.

```c
#include <stdio.h>


int main() {
  // /* This block of code is commented out
  // int x = 10;
  // printf("Value of x: %d\n", x);
  // */


  printf("Program runs without errors.\n");
  return 0;
}
```

Output

```
Program runs without errors.
```

Best Practices for Writing Comments in C

Following are some best practices to write comments in your C program:

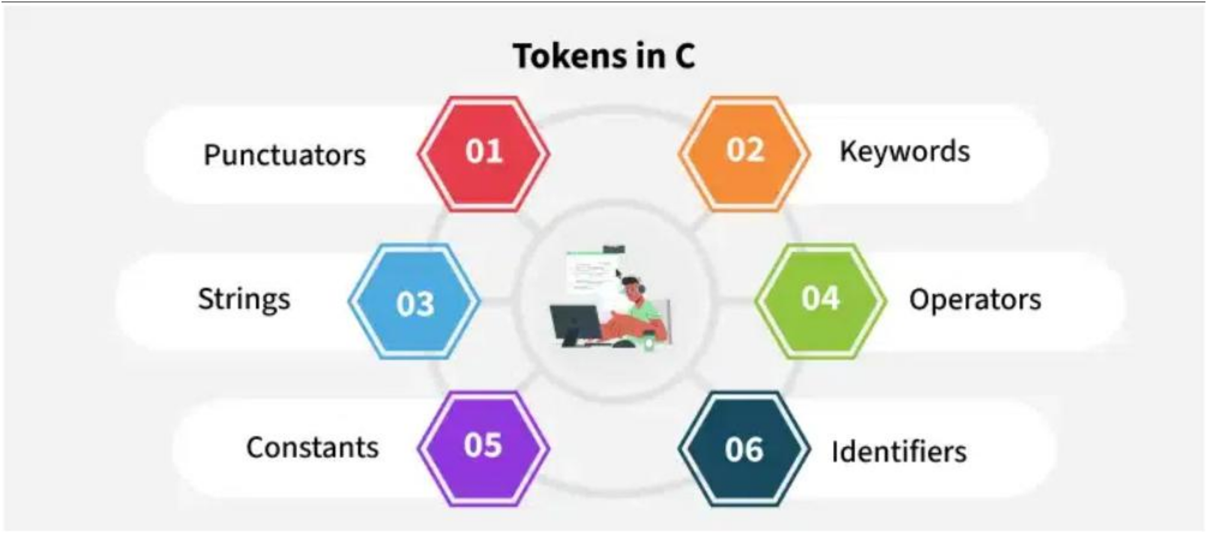
- Write comments that are easy to understand.
- Do not comment on every line unnecessarily and avoid writing obvious comments.
- Update comments regularly.
- Focus on explaining the intent behind the code rather than restating the obvious.

Tokens in C

Last Updated: 23 Jul, 2025

In C programming, tokens are the smallest units in a program that have meaningful representations. Tokens are the building blocks of a C program, and they are recognized by the C compiler to form valid expressions and statements. Tokens can be classified into various categories, each with specific roles in the 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:

Table of Content

1. Punctuators

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose. Some of these are listed below:

- **Brackets[]:** Opening and closing brackets are used as array element references. These indicate single and multidimensional subscripts.
- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Braces{}:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.

- **Comma (,):** It is used to separate more than one statement like for separating parameters in function calls.
- **Colon(:):** It is an operator that essentially invokes something called an initialization list.
- **Semicolon(;):** It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- **Asterisk (*):** It is used to create a pointer variable and for the multiplication of variables.
- **Assignment operator(=):** It is used to assign values and for logical operation validation.
- **Pre-processor (#):** The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.
- **Dot (.):** Used to access members of a structure or union.
- **Tilde(~):** Bitwise One's Complement Operator.

Example:

```
#include <stdio.h>


int main() {


    // '\n' is a special symbol that
    // represents a newline
    printf("Hello, \n World!");
    return 0;
}
```

Output

```
Hello,
 World!
```

2. Keywords

Keywords are reserved words that have predefined meanings in C. These cannot be used as identifiers (variable names, function names, etc.). Keywords define the structure and behavior of the program **C** language supports **32** keywords such as int, for, if, ... etc.

Example:

```
#include <stdio.h>


int main() {


    // 'int' is a keyword used to define
    // variable type
    int x = 5;
    printf("%d", x);


    // 'return' is a keyword used to exit
    // main function
    return 0;
}
```

Output

```
5
```

Note: *The number of keywords may change depending on the version of C you are using. For example, keywords present in ANSI C are 32 while in C11, it was increased to 44. Moreover, in the latest c23, it is increased to around 54.*

3. Strings

Strings are nothing but an array of characters ended with a null character (‘\0’). This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas a character is enclosed in single quotes in C and C++.

Examples:

```
#include <stdio.h>


int main() {


    // "Hello, World!" is a string literal
    char str[] = "Hello, World!";
    printf("%s", str);
    return 0;
}
```

Output

```
Hello, World!
```

4. Operators

Operators are symbols that trigger an action when applied to C variables and other objects. The data items on which operators act are called operands.

Example:

```
#include <stdio.h>


int main() {

   int a = 10, b = 5;


   // '+' is an arithmetic operator used

   // for addition

   int sum = a + b;

   printf("%d", sum);

   return 0;

}
```

Output

```
15
```

5. Identifiers

Identifiers are names given to variables, functions, arrays, and other user-defined items. They must begin with a letter (a-z, A-Z) or an underscore (_) and can be followed by letters, digits (0-9), and underscores.

Example:

```
#include <stdio.h>


int main() {


   // 'num' is an identifier used to name

   // a variable

   int num = 10;
```

```
    printf("%d", num);
    return 0;
}
```

Output

```
10
```

6. Constants

Constants are fixed values used in a C program. These values do not change during the execution of the program. Constants can be integers, floating-point numbers, characters, or strings.

Examples:

```
#include <stdio.h>

int main() {

    // 'MAX_VALUE' is a constant that holds
    // a fixed value
    const int MAX_VALUE = 100;
    printf("%d", MAX_VALUE);
    return 0;
}
```

Output

```
100
```

Keywords in C

Last Updated: 13 Jul, 2025

Keywords are predefined or reserved words that have special meanings to the compiler. These are part of the syntax and cannot be used as identifiers in the program. A list of keywords in C or reserved words in the C programming language are mentioned 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

We cannot use these keywords as identifiers (such as variable names, function names, or struct names). The compiler will throw an **error** if we try to do so.

Example:

```
#include <stdio.h>


int main() {
   int return = 10;
   printf("%d\n", return);
   return 0;
}
```

Output

```
./Solution.c: In function 'main':
./Solution.c:4:9: error: expected identifier or '(' before 'return'
    int return = 10;
        ^
./Solution.c:5:20: error: expected expression before 'return'
    printf("%d\n", return);
                   ^
```

Let's categorize all keywords based on context for a more clear understanding.

Category	**Keywords**
Data Type Keywords	char, int, float, double, void, short, long, signed, unsigned
Operator & Utility Keywords	sizeof, return, goto, typedef

Category	Keywords
Control Flow Keywords	if, else, switch, case, default, for, while, do, break, continue
Storage Class Keywords	auto, register, static, extern
Type Qualifiers	const, volatile
User Defined Types	struct, union, enum

Difference Between Keywords and Identifiers

Keywords	Identifiers
Reserved Words in C that have a specific meaning and use in the syntax	Names given to variables, functions, structs, etc.
Cannot be used as variable names.	Can be used as variable names (if not a keyword).
Examples: int, return, if, while	Examples: x, total, count
Part of the C language grammar.	User-defined, meaningful names in the code.
Cannot be redefined or repurposed.	Can be defined redefined and reused as needed.

Difference Between Variables and Keywords

Variables	Keywords
Used to store data and manipulate data	Reserved words used to define structure and control flow

Variables	Keywords
Created by the user using valid identifiers	Defined by C standard and recognized by the compiler
Examples: age, total, marks	Examples: for, while, int, return
Can hold values and be used in operations	Used to define the structure of C code.
Naming is flexible within identifier rules	Names are fixed and cannot be altered or used differently