



WIKIPEDIA
The Free Encyclopedia

C (programming language)



C (*pronounced* /ˈsiː/ – *like the letter c*)^[6] is a general-purpose computer programming language. It was created in the 1970s by Dennis Ritchie, and remains very widely used and influential. By design, C's features cleanly reflect the capabilities of the targeted CPUs. It has found lasting use in operating systems, device drivers, and protocol stacks, but its use in application software has been decreasing.^[7] C is commonly used on computer architectures that range from the largest supercomputers to the smallest microcontrollers and embedded systems.

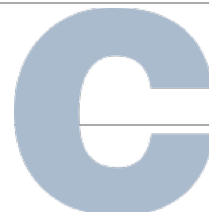
A successor to the programming language B, C was originally developed at Bell Labs by Ritchie between 1972 and 1973 to construct utilities running on Unix. It was applied to re-implementing the kernel of the Unix operating system.^[8] During the 1980s, C gradually gained popularity. It has become one of the most widely used programming languages,^{[9][10]} with C compilers available for practically all modern computer architectures and operating systems. The book *The C Programming Language*, co-authored by the original language designer, served for many years as the *de facto* standard for the language.^{[11][12]} C has been standardized since 1989 by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO).

C is an imperative procedural language, supporting structured programming, lexical variable scope, and recursion, with a static type system. It was designed to be compiled to provide low-level access to memory and language constructs that map efficiently to machine instructions, all with minimal runtime support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant C program written with portability in mind can be compiled for a wide variety of computer platforms and operating systems with few changes to its source code.^[13]

Since 2000, C has consistently ranked among the top two languages in the TIOBE index, a measure of the popularity of programming languages.^[14]

Overview

C



Cover graphic of *The C Programming Language* book.^[1]


Paradigm	Multi-paradigm : <div>imperative<div></div>(procedural),<div></div>structured</div>
Designed by	Dennis Ritchie
Developer	ANSI X3J11 <div></div> (ANSI C) ; <div></div> ISO/IEC JTC 1 <div></div> (Joint Technical Committee 1) / SC 22 <div></div> (Subcommittee 22) / WG 14 <div></div> (Working Group 14) (ISO C)
First appeared	1972 ^[2]
Stable release	C17 / June 2018
Preview release	C23 (N3096 (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf)) / April 2, 2023 ^[3]
Typing discipline	Static , weak , <div></div> manifest , nominal
OS	Cross-platform

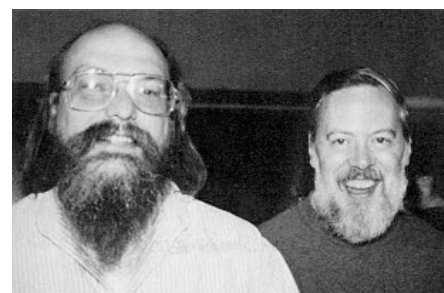
C is an imperative, procedural language in the ALGOL tradition. It has a static type system. In C, all executable code is contained within subroutines (also called "functions", though not in the sense of functional programming). Function parameters are passed by value, although arrays are passed as pointers, i.e. the address of the first item in the array. *Pass-by-reference* is simulated in C by explicitly passing pointers to the thing being referenced.

C program source text is free-form code. Semicolons terminate statements, while curly braces are used to group statements into blocks.

The C language also exhibits the following characteristics:

- The language has a small, fixed number of keywords, including a full set of control flow primitives: if/else, for, do/while, while, and switch. User-defined names are not distinguished from keywords by any kind of sigil.
- It has a large number of arithmetic, bitwise, and logic operators: +, +=, ++, &, |, etc.
- More than one assignment may be performed in a single statement.
- Functions:
 - Function return values can be ignored, when not needed.
 - Function and data pointers permit *ad hoc* run-time polymorphism.
 - Functions may not be defined within the lexical scope of other functions.
 - Variables may be defined within a function, with scope.
 - A function may call itself, so recursion is supported.
- Data typing is static, but weakly enforced; all data has a type, but implicit conversions are possible.
- User-defined (typedef) and compound types are possible.
 - Heterogeneous aggregate data types (struct) allow related data elements to be accessed and assigned as a unit. The contents of whole structs cannot be compared using a single built-in operator (the elements must be compared individually).
 - Union is a structure with overlapping members; it allows multiple data types to share the same memory location.
 - Array indexing is a secondary notation, defined in terms of pointer arithmetic. Whole arrays cannot be assigned or compared using a single built-in operator. There is no "array" keyword in use or definition; instead, square brackets indicate arrays syntactically, for example `month[11]`.

Filename extensions	.c, .h
Website	<u>www.iso.org/standard/74528.html</u> (<u>https://www.iso.org/standard/74528.html</u>) <u>www.open-std.org/jtc1/sc22/wg14/</u> (<u>http://www.open-std.org/jtc1/sc22/wg14/</u>)
Major implementations	<u>pcc</u> , <u>GCC</u> , <u>Clang</u> , <u>Intel C</u> , <u>C++Builder</u> , <u>Microsoft Visual C++</u> , <u>Watcom C</u>
Dialects	<u>Cyclone</u> , <u>Unified Parallel C</u> , <u>Split-C</u> , <u>Cilk</u> , <u>C*</u>
Influenced by	<u>B</u> (<u>BCPL</u> , <u>CPL</u>), <u>ALGOL 68</u> , ^[4] <u>PL/I</u> , <u>FORTRAN</u>
Influenced	Numerous: <u>AMPL</u> , <u>AWK</u> , <u>csh</u> , <u>C++</u> , <u>C--</u> , <u>C#</u> , <u>Objective-C</u> , <u>D</u> , <u>Go</u> , <u>Java</u> , <u>JavaScript</u> , <u>JS++</u> , <u>Julia</u> , <u>Limbo</u> , <u>LPC</u> , <u>Perl</u> , <u>PHP</u> , <u>Pike</u> , <u>Processing</u> , <u>Python</u> , <u>Rust</u> , <u>Seed7</u> , <u>Vala</u> , <u>Verilog (HDL)</u> , ^[5] <u>Nim</u> , <u>Zig</u>
 C Programming at Wikibooks	



Dennis Ritchie (right), the inventor of the C programming language, with Ken Thompson

- Enumerated types are possible with the `enum` keyword. They are freely interconvertible with integers.
- Strings are not a distinct data type, but are conventionally implemented as null-terminated character arrays.
- Low-level access to computer memory is possible by converting machine addresses to pointers.
- Procedures (subroutines not returning values) are a special case of function, with an empty return type `void`.
- Memory can be allocated to a program with calls to library routines.
- A preprocessor performs macro definition, source code file inclusion, and conditional compilation.
- There is a basic form of modularity: files can be compiled separately and linked together, with control over which functions and data objects are visible to other files via static and extern attributes.
- Complex functionality such as I/O, string manipulation, and mathematical functions are consistently delegated to library routines.
- The generated code after compilation has relatively straightforward needs on the underlying platform, which makes it suitable for creating operating systems and for use in embedded systems.

While C does not include certain features found in other languages (such as object orientation and garbage collection), these can be implemented or emulated, often through the use of external libraries (e.g., the GLib Object System or the Boehm garbage collector).

Relations to other languages

Many later languages have borrowed directly or indirectly from C, including C++, C#, Unix's C shell, D, Go, Java, JavaScript (including transpilers), Julia, Limbo, LPC, Objective-C, Perl, PHP, Python, Ruby, Rust, Swift, Verilog and SystemVerilog (hardware description languages).^[5] These languages have drawn many of their control structures and other basic features from C. Most of them (Python being a dramatic exception) also express highly similar syntax to C, and they tend to combine the recognizable expression and statement syntax of C with underlying type systems, data models, and semantics that can be radically different.

History

Early developments

The origin of C is closely tied to the development of the Unix operating system, originally implemented in assembly language on a PDP-7 by Dennis Ritchie and Ken Thompson, incorporating several ideas from colleagues. Eventually, they decided to port the operating system to a PDP-11. The original PDP-11 version of Unix was also developed in assembly language.^[8]

B

Thompson wanted a programming language for developing utilities for the new platform. At first, he tried to write a Fortran compiler, but soon gave up the idea. Instead, he created a cut-down version of the recently developed BCPL systems programming language. The official description of BCPL was not available at the time^[15] and Thompson modified the syntax to be less wordy, and similar to a simplified ALGOL known as SMALGOL.^[16] Thompson called the result B.^[8] He described B as "BCPL semantics with a lot of SMALGOL syntax".^[16] Like BCPL, B had a bootstrapping compiler to facilitate porting to new machines.^[16] However, few utilities were ultimately written in B because it was too slow, and could not take advantage of PDP-11 features such as byte addressability.

Timeline of C language^[13]

Year	Informal name	C Standard
1972	Birth	—
1978	<u>K&R C</u>	—
1989/1990	<u>ANSI C</u> , <u>ISO C</u>	ISO/IEC 9899:1990
1999	<u>C99</u>	ISO/IEC 9899:1999
2011	<u>C11</u> , C1x	ISO/IEC 9899:2011
2018	<u>C17</u>	ISO/IEC 9899:2018
2024*	<u>C23</u> , C2x	ISO/IEC 9899:2023

New B and first C release

In 1971, Ritchie started to improve B, to utilise the features of the more-powerful PDP-11. A significant addition was a character data type. He called this *New B* (NB).^[16] Thompson started to use NB to write the Unix kernel, and his requirements shaped the direction of the language development.^{[16][17]} Through to 1972, richer types were added to the NB language: NB had arrays of `int` and `char`. Pointers, the ability to generate pointers to other types, arrays of all types, and types to be returned from functions were all also added. Arrays within expressions became pointers. A new compiler was written, and the language was renamed C.^[8]

The C compiler and some utilities made with it were included in Version 2 Unix, which is also known as Research Unix.^[18]

Structures and the Unix kernel re-write

At Version 4 Unix, released in November 1973, the Unix kernel was extensively re-implemented in C.^[8] By this time, the C language had acquired some powerful features such as `struct` types.

The preprocessor was introduced around 1973 at the urging of Alan Snyder and also in recognition of the usefulness of the file-inclusion mechanisms available in BCPL and PL/I. Its original version provided only included files and simple string replacements: `#include` and `#define` of parameterless macros. Soon after that, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation.^[8]

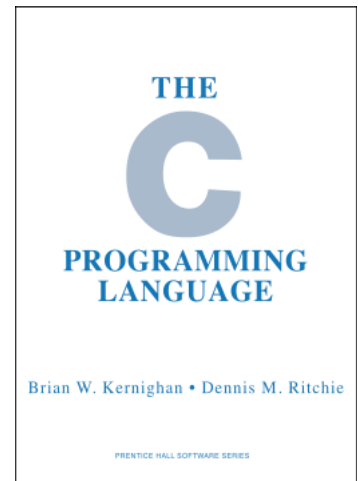
Unix was one of the first operating system kernels implemented in a language other than assembly. Earlier instances include the Multics system (which was written in PL/I) and Master Control Program (MCP) for the Burroughs B5000 (which was written in ALGOL) in 1961. In around 1977, Ritchie and Stephen C. Johnson made further changes to the language to facilitate portability of the Unix operating system. Johnson's Portable C Compiler served as the basis for several implementations of C on new platforms.^[17]

K&R C

In 1978, [Brian Kernighan](#) and [Dennis Ritchie](#) published the first edition of *The C Programming Language*.^[19] Known as *K&R* from the initials of its authors, the book served for many years as an informal specification of the language. The version of C that it describes is commonly referred to as "**K&R C**". As this was released in 1978, it is also referred to as C78.^[20] The second edition of the book^[21] covers the later ANSI C standard, described below.

K&R introduced several language features:

- Standard I/O library
- long int data type
- unsigned int data type
- Compound assignment operators of the form *=op* (such as *=-*) were changed to the form *op=* (that is, *-=*) to remove the semantic ambiguity created by constructs such as *i=-10*, which had been interpreted as *i -= 10* (decrement *i* by 10) instead of the possibly intended *i = -10* (let *i* be *-10*).



The cover of the book *The C Programming Language*, first edition, by [Brian Kernighan](#) and [Dennis Ritchie](#)

Even after the publication of the 1989 ANSI standard, for many years K&R C was still considered the "lowest common denominator" to which C programmers restricted themselves when maximum portability was desired, since many older compilers were still in use, and because carefully written K&R C code can be legal Standard C as well.

In early versions of C, only functions that return types other than `int` must be declared if used before the function definition; functions used without prior declaration were presumed to return type `int`.

For example:

```
long some_function(); /* This is a function declaration, so the compiler can know the name and return type of this
function. */
/* int */ other_function(); /* Another function declaration. Because this is an early version of C, there is an
implicit 'int' type here. A comment shows where the explicit 'int' type specifier would be required in later
versions. */

/* int */ calling_function() /* This is a function definition, including the body of the code following in the {
curly brackets }. Because no return type is specified, the function implicitly returns an 'int' in this early
version of C. */
{
    long test1;
    register /* int */ test2; /* Again, note that 'int' is not required here. The 'int' type specifier */
                             /* in the comment would be required in later versions of C. */
                             /* The 'register' keyword indicates to the compiler that this variable should */
                             /* ideally be stored in a register as opposed to within the stack frame. */

    test1 = some_function();
    if (test1 > 1)
        test2 = 0;
    else
        test2 = other_function();
    return test2;
}
```

The `int` type specifiers which are commented out could be omitted in K&R C, but are required in later standards.

Since K&R function declarations did not include any information about function arguments, function parameter type checks were not performed, although some compilers would issue a warning message if a local function was called with the wrong number of arguments, or if multiple calls to an external function used different numbers or types of arguments. Separate tools such as Unix's lint utility were developed that (among other things) could check for consistency of function use across multiple source files.

In the years following the publication of K&R C, several features were added to the language, supported by compilers from AT&T (in particular PCC^[22]) and some other vendors. These included:

- void functions (i.e., functions with no return value)
- functions returning struct or union types (previously only a single pointer, integer or float could be returned)
- assignment for struct data types
- enumerated types (previously, preprocessor definitions for integer fixed values were used, e.g. `#define GREEN 3`)

The large number of extensions and lack of agreement on a standard library, together with the language popularity and the fact that not even the Unix compilers precisely implemented the K&R specification, led to the necessity of standardization.

ANSI C and ISO C

During the late 1970s and 1980s, versions of C were implemented for a wide variety of mainframe computers, minicomputers, and microcomputers, including the IBM PC, as its popularity began to increase significantly.

In 1983, the American National Standards Institute (ANSI) formed a committee, X3J11, to establish a standard specification of C. X3J11 based the C standard on the Unix implementation; however, the non-portable portion of the Unix C library was handed off to the IEEE working group 1003 to become the basis for the 1988 POSIX standard. In 1989, the C standard was ratified as ANSI X3.159-1989 "Programming Language C". This version of the language is often referred to as ANSI C, Standard C, or sometimes C89.

In 1990, the ANSI C standard (with formatting changes) was adopted by the International Organization for Standardization (ISO) as ISO/IEC 9899:1990, which is sometimes called C90. Therefore, the terms "C89" and "C90" refer to the same programming language.

ANSI, like other national standards bodies, no longer develops the C standard independently, but defers to the international C standard, maintained by the working group ISO/IEC JTC1/SC22 /WG14. National adoption of an update to the international standard typically occurs within a year of ISO publication.

One of the aims of the C standardization process was to produce a superset of K&R C, incorporating many of the subsequently introduced unofficial features. The standards committee

also included several additional features such as function prototypes (borrowed from C++), void pointers, support for international character sets and locales, and preprocessor enhancements. Although the syntax for parameter declarations was augmented to include the style used in C++, the K&R interface continued to be permitted, for compatibility with existing source code.

C89 is supported by current C compilers, and most modern C code is based on it. Any program written only in Standard C and without any hardware-dependent assumptions will run correctly on any platform with a conforming C implementation, within its resource limits. Without such precautions, programs may compile only on a certain platform or with a particular compiler, due, for example, to the use of non-standard libraries, such as GUI libraries, or to a reliance on compiler- or platform-specific attributes such as the exact size of data types and byte endianness.

In cases where code must be compilable by either standard-conforming or K&R C-based compilers, the `__STDC__` macro can be used to split the code into Standard and K&R sections to prevent the use on a K&R C-based compiler of features available only in Standard C.

After the ANSI/ISO standardization process, the C language specification remained relatively static for several years. In 1995, Normative Amendment 1 to the 1990 C standard (ISO/IEC 9899/AMD1:1995, known informally as C95) was published, to correct some details and to add more extensive support for international character sets.^[23]

C99

The C standard was further revised in the late 1990s, leading to the publication of ISO/IEC 9899:1999 in 1999, which is commonly referred to as "C99". It has since been amended three times by Technical Corrigenda.^[24]

C99 introduced several new features, including inline functions, several new data types (including `long long int` and a complex type to represent complex numbers), variable-length arrays and flexible array members, improved support for IEEE 754 floating point, support for variadic macros (macros of variable arity), and support for one-line comments beginning with `//`, as in BCPL or C++. Many of these had already been implemented as extensions in several C compilers.

C99 is for the most part backward compatible with C90, but is stricter in some ways; in particular, a declaration that lacks a type specifier no longer has `int` implicitly assumed. A standard macro `__STDC_VERSION__` is defined with value 199901L to indicate that C99 support is available. GCC, Solaris Studio, and other C compilers now support many or all of the new features of C99. The C compiler in Microsoft Visual C++, however, implements the C89 standard and those parts of C99 that are required for compatibility with C++11.^[25]

In addition, the C99 standard requires support for Unicode identifiers in the form of escaped characters (e.g. `\u0040` or `\U0001f431`) and suggests support for raw Unicode names.

C11

In 2007, work began on another revision of the C standard, informally called "C1X" until its official publication of ISO/IEC 9899:2011 on 2011-12-08. The C standards committee adopted guidelines to limit the adoption of new features that had not been tested by existing implementations.

The C11 standard adds numerous new features to C and the library, including type generic macros, anonymous structures, improved Unicode support, atomic operations, multi-threading, and bounds-checked functions. It also makes some portions of the existing C99 library optional, and improves compatibility with C++. The standard macro `__STDC_VERSION__` is defined as 201112L to indicate that C11 support is available.

C17

Published in June 2018 as ISO/IEC 9899:2018, C17 is the current standard for the C programming language. It introduces no new language features, only technical corrections, and clarifications to defects in C11. The standard macro `__STDC_VERSION__` is defined as 201710L.

C23

C23 is the informal name for the next (after C17) major C language standard revision. It is expected to be published in 2024.^[26]

Embedded C

Historically, embedded C programming requires nonstandard extensions to the C language in order to support exotic features such as fixed-point arithmetic, multiple distinct memory banks, and basic I/O operations.

In 2008, the C Standards Committee published a technical report extending the C language^[27] to address these issues by providing a common standard for all implementations to adhere to. It includes a number of features not available in normal C, such as fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

Syntax

C has a formal grammar specified by the C standard.^[28] Line endings are generally not significant in C; however, line boundaries do have significance during the preprocessing phase. Comments may appear either between the delimiters `/*` and `*/`, or (since C99) following `//` until the end of the line. Comments delimited by `/*` and `*/` do not nest, and these sequences of characters are not interpreted as comment delimiters if they appear inside string or character literals.^[29]

C source files contain declarations and function definitions. Function definitions, in turn, contain declarations and statements. Declarations either define new types using keywords such as `struct`, `union`, and `enum`, or assign types to and perhaps reserve storage for new variables, usually by writing the type followed by the variable name. Keywords such as `char` and `int` specify built-in types. Sections of code are enclosed in braces (`{` and `}`, sometimes called "curly brackets") to limit the scope of declarations and to act as a single statement for control structures.

As an imperative language, C uses *statements* to specify actions. The most common statement is an *expression statement*, consisting of an expression to be evaluated, followed by a semicolon; as a side effect of the evaluation, functions may be called and variables may be assigned new values. To modify the normal sequential execution of statements, C provides several control-flow statements

identified by reserved keywords. Structured programming is supported by `if ... [else]` conditional execution and by `do ... while`, `while`, and `for` iterative execution (looping). The `for` statement has separate initialization, testing, and reinitialization expressions, any or all of which can be omitted. `break` and `continue` can be used within the loop. `Break` is used to leave the innermost enclosing loop statement and `continue` is used to skip to its reinitialisation. There is also a non-structured `goto` statement which branches directly to the designated label within the function. `switch` selects a case to be executed based on the value of an integer expression. Different from many other languages, control-flow will fall through to the next case unless terminated by a `break`.

Expressions can use a variety of built-in operators and may contain function calls. The order in which arguments to functions and operands to most operators are evaluated is unspecified. The evaluations may even be interleaved. However, all side effects (including storage to variables) will occur before the next "sequence point"; sequence points include the end of each expression statement, and the entry to and return from each function call. Sequence points also occur during evaluation of expressions containing certain operators (`&&`, `||`, `?:` and the comma operator). This permits a high degree of object code optimization by the compiler, but requires C programmers to take more care to obtain reliable results than is needed for other programming languages.

Kernighan and Ritchie say in the Introduction of *The C Programming Language*: "C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better."^[30] The C standard did not attempt to correct many of these blemishes, because of the impact of such changes on already existing software.

Character set

The basic C source character set includes the following characters:

- Lowercase and uppercase letters of ISO Basic Latin Alphabet: `a–z A–Z`
- Decimal digits: `0–9`
- Graphic characters: `! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~`
- Whitespace characters: *space*, *horizontal tab*, *vertical tab*, *form feed*, *newline*

Newline indicates the end of a text line; it need not correspond to an actual single character, although for convenience C treats it as one.

Additional multi-byte encoded characters may be used in string literals, but they are not entirely portable. The latest C standard (C11) allows multi-national Unicode characters to be embedded portably within C source text by using `\uXXXX` or `\UXXXXXXXX` encoding (where the X denotes a hexadecimal character), although this feature is not yet widely implemented.

The basic C execution character set contains the same characters, along with representations for alert, backspace, and carriage return. Run-time support for extended character sets has increased with each revision of the C standard.

Reserved words

C89 has 32 reserved words, also known as keywords, which are the words that cannot be used for any purposes other than those for which they are predefined:

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

C99 reserved five more words:

- | | | |
|-------------------|---------------------|-------------------|
| ▪ <u>_Bool</u> | ▪ <u>_Imaginary</u> | ▪ <u>restrict</u> |
| ▪ <u>_Complex</u> | ▪ <u>inline</u> | |

C11 reserved seven more words:^[31]

- | | | |
|-------------------|-------------------------|------------------------|
| ▪ <u>_Alignas</u> | ▪ <u>_Generic</u> | ▪ <u>_Thread_local</u> |
| ▪ <u>_Alignof</u> | ▪ <u>_Noreturn</u> | |
| ▪ <u>_Atomic</u> | ▪ <u>_Static_assert</u> | |

C23 will reserve 14 more words:

- | | | |
|--------------------|------------------------|------------------------|
| ▪ <u>alignas</u> | ▪ <u>nullptr</u> | ▪ <u>typeof_unqual</u> |
| ▪ <u>alignof</u> | ▪ <u>static_assert</u> | ▪ <u>_Decimal128</u> |
| ▪ <u>bool</u> | ▪ <u>thread_local</u> | ▪ <u>_Decimal32</u> |
| ▪ <u>constexpr</u> | ▪ <u>true</u> | ▪ <u>_Decimal64</u> |
| ▪ <u>false</u> | ▪ <u>typeof</u> | |

Most of the recently reserved words begin with an underscore followed by a capital letter, because identifiers of that form were previously reserved by the C standard for use only by implementations. Since existing program source code should not have been using these identifiers, it would not be affected when C implementations started supporting these extensions to the programming language. Some standard headers do define more convenient synonyms for underscored identifiers. Some of those words were added as keywords with their conventional spelling in C23 and the corresponding macros were removed. The language previously included a reserved word called entry, but this was seldom implemented, and has now been removed as a reserved word.^[32]

Operators

C supports a rich set of operators, which are symbols used within an expression to specify the manipulations to be performed while evaluating that expression. C has operators for:

- arithmetic: `+`, `-`, `*`, `/`, `%`
- assignment: `=`
- augmented assignment: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- bitwise logic: `~`, `&`, `|`, `^`
- bitwise shifts: `<<`, `>>`
- Boolean logic: `!`, `&&`, `||`
- conditional evaluation: `?` `:`
- equality testing: `==`, `!=`
- calling functions: `()`
- increment and decrement: `++`, `--`
- member selection: `.`, `->`
- object size: `sizeof`
- type: `typeof`, `typeof_unqual` *since C23*
- order relations: `<`, `<=`, `>`, `>=`
- reference and dereference: `&`, `*`, `[]`
- sequencing: `,`
- subexpression grouping: `()`
- type conversion: `(typename)`

C uses the operator `=` (used in mathematics to express equality) to indicate assignment, following the precedent of Fortran and PL/I, but unlike ALGOL and its derivatives. C uses the operator `==` to test for equality. The similarity between these two operators (assignment and equality) may result in the accidental use of one in place of the other, and in many cases, the mistake does not produce an error message (although some compilers produce warnings). For example, the conditional expression `if (a == b + 1)` might mistakenly be written as `if (a = b + 1)`, which will be evaluated as true if `a` is not zero after the assignment.^[33]

The C operator precedence is not always intuitive. For example, the operator `==` binds more tightly than (is executed prior to) the operators `&` (bitwise AND) and `|` (bitwise OR) in expressions such as `x & 1 == 0`, which must be written as `(x & 1) == 0` if that is the coder's intent.^[34]

"Hello, world" example

The "hello, world" example, which appeared in the first edition of *K&R*, has become the model for an introductory program in most programming textbooks. The program prints "hello, world" to the standard output, which is usually a terminal or screen display.

The original version was:^[35]

```
main()
{
    printf("hello, world\n");
}
```

A standard-conforming "hello, world" program is:^[a]

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

The first line of the program contains a preprocessing directive, indicated by `#include`. This causes the compiler to replace that line with the entire text of the `stdio.h` standard header, which contains declarations for standard input and output functions such as `printf` and `scanf`. The angle brackets surrounding `stdio.h` indicate that `stdio.h` can be located using a search strategy that prefers headers provided with the compiler to other headers having the same name, as opposed to double quotes which typically include local or project-specific header files.

The next line indicates that a function named `main` is being defined. The `main` function serves a special purpose in C programs; the run-time environment calls the `main` function to begin program execution. The type specifier `int` indicates that the value that is returned to the invoker (in this case the run-time environment) as a result of evaluating the `main` function, is an integer. The keyword `void` as a parameter list indicates that this function takes no arguments.^[b]

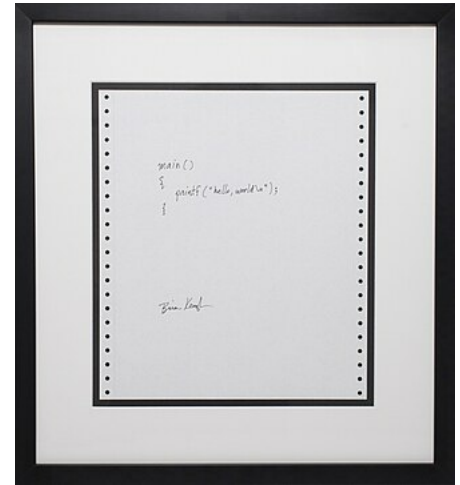
The opening curly brace indicates the beginning of the definition of the `main` function.

The next line *calls* (diverts execution to) a function named `printf`, which in this case is supplied from a system library. In this call, the `printf` function is *passed* (provided with) a single argument, the address of the first character in the string literal `"hello, world\n"`. The string literal is an unnamed array with elements of type `char`, set up automatically by the compiler with a final `NULL` (ASCII value 0) character to mark the end of the array (for `printf` to know the length of the string). The `NULL` character can be also written as an escape sequence, written as `\0`. The `\n` is an *escape sequence* that C translates to a *newline* character, which on output signifies the end of the current line. The return value of the `printf` function is of type `int`, but it is silently discarded since it is not used. (A more careful program might test the return value to determine whether or not the `printf` function succeeded.) The semicolon `;` terminates the statement.

The closing curly brace indicates the end of the code for the `main` function. According to the C99 specification and newer, the `main` function, unlike any other function, will implicitly return a value of 0 upon reaching the `}` that terminates the function. (Formerly an explicit `return 0;` statement was required.) This is interpreted by the run-time system as an exit code indicating successful execution.^[36]

Data types

The type system in C is static and weakly typed, which makes it similar to the type system of ALGOL descendants such as Pascal.^[37] There are built-in types for integers of various sizes, both signed and unsigned, floating-point numbers, and enumerated types (`enum`). Integer type `char` is often used for single-byte characters. C99 added a Boolean datatype. There are also derived types including arrays, pointers, records (`struct`), and unions (`union`).



"Hello, World!" program by Brian Kernighan (1978)

C is often used in low-level systems programming where escapes from the type system may be necessary. The compiler attempts to ensure type correctness of most expressions, but the programmer can override the checks in various ways, either by using a *type cast* to explicitly convert a value from one type to another, or by using pointers or unions to reinterpret the underlying bits of a data object in some other way.

Some find C's declaration syntax unintuitive, particularly for function pointers. (Ritchie's idea was to declare identifiers in contexts resembling their use: "declaration reflects use".)[38]

C's *usual arithmetic conversions* allow for efficient code to be generated, but can sometimes produce unexpected results. For example, a comparison of signed and unsigned integers of equal width requires a conversion of the signed value to unsigned. This can generate unexpected results if the signed value is negative.

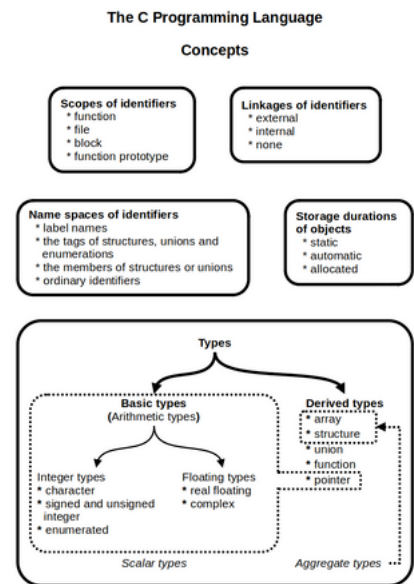
Pointers

C supports the use of pointers, a type of reference that records the address or location of an object or function in memory. Pointers can be *dereferenced* to access data stored at the address pointed to, or to invoke a pointed-to function. Pointers can be manipulated using assignment or pointer arithmetic. The run-time representation of a pointer value is typically a raw memory address (perhaps augmented by an offset-within-word field), but since a pointer's type includes the type of the thing pointed to, expressions including pointers can be type-checked at compile time. Pointer arithmetic is automatically scaled by the size of the pointed-to data type.

Pointers are used for many purposes in C. Text strings are commonly manipulated using pointers into arrays of characters. Dynamic memory allocation is performed using pointers; the result of a `malloc` is usually cast to the data type of the data to be stored. Many data types, such as trees, are commonly implemented as dynamically allocated struct objects linked together using pointers. Pointers to other pointers are often used in multi-dimensional arrays and arrays of struct objects. Pointers to functions (*function pointers*) are useful for passing functions as arguments to higher-order functions (such as `qsort` or `bsearch`), in dispatch tables, or as callbacks to event handlers. [36]

A *null pointer value* explicitly points to no valid location. Dereferencing a null pointer value is undefined, often resulting in a segmentation fault. Null pointer values are useful for indicating special cases such as no "next" pointer in the final node of a linked list, or as an error indication from functions returning pointers. In appropriate contexts in source code, such as for assigning to a pointer variable, a *null pointer constant* can be written as `0`, with or without explicit casting to a pointer type, as the `NULL` macro defined by several standard headers or, since C23 with the constant `nullptr`. In conditional contexts, null pointer values evaluate to false, while all other pointer values evaluate to true.

Void pointers (`void *`) point to objects of unspecified type, and can therefore be used as "generic" data pointers. Since the size and type of the pointed-to object is not known, void pointers cannot be dereferenced, nor is pointer arithmetic on them allowed, although they can easily be (and in many



contexts implicitly are) converted to and from any other object pointer type.^[36]

Careless use of pointers is potentially dangerous. Because they are typically unchecked, a pointer variable can be made to point to any arbitrary location, which can cause undesirable effects. Although properly used pointers point to safe places, they can be made to point to unsafe places by using invalid pointer arithmetic; the objects they point to may continue to be used after deallocation (dangling pointers); they may be used without having been initialized (wild pointers); or they may be directly assigned an unsafe value using a cast, union, or through another corrupt pointer. In general, C is permissive in allowing manipulation of and conversion between pointer types, although compilers typically provide options for various levels of checking. Some other programming languages address these problems by using more restrictive reference types.

Arrays

Array types in C are traditionally of a fixed, static size specified at compile time. The more recent C99 standard also allows a form of variable-length arrays. However, it is also possible to allocate a block of memory (of arbitrary size) at run-time, using the standard library's `malloc` function, and treat it as an array.

Since arrays are always accessed (in effect) via pointers, array accesses are typically *not* checked against the underlying array size, although some compilers may provide bounds checking as an option.^{[39][40]} Array bounds violations are therefore possible and can lead to various repercussions, including illegal memory accesses, corruption of data, buffer overruns, and run-time exceptions.

C does not have a special provision for declaring multi-dimensional arrays, but rather relies on recursion within the type system to declare arrays of arrays, which effectively accomplishes the same thing. The index values of the resulting "multi-dimensional array" can be thought of as increasing in row-major order. Multi-dimensional arrays are commonly used in numerical algorithms (mainly from applied linear algebra) to store matrices. The structure of the C array is well suited to this particular task. However, in early versions of C the bounds of the array must be known fixed values or else explicitly passed to any subroutine that requires them, and dynamically sized arrays of arrays cannot be accessed using double indexing. (A workaround for this was to allocate the array with an additional "row vector" of pointers to the columns.) C99 introduced "variable-length arrays" which address this issue.

The following example using modern C (C99 or later) shows allocation of a two-dimensional array on the heap and the use of multi-dimensional array indexing for accesses (which can use bounds-checking on many C compilers):

```
int func(int N, int M)
{
    float (*p)[N][M] = malloc(sizeof *p);
    if (!p)
        return -1;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            (*p)[i][j] = i + j;
    print_array(N, M, p);
    free(p);
    return 1;
}
```

And here is a similar implementation using C99's Auto VLA feature:

```
int func(int N, int M)
{
    // Caution: checks should be made to ensure N*M*sizeof(float) does NOT exceed limitations for auto VLAs and is
    // within available size of stack.
    float p[N][M]; // auto VLA is held on the stack, and sized when the function is invoked
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            p[i][j] = i + j;
    // no need to free(p) since it will disappear when the function exits, along with the rest of the stack frame
    return 1;
}
```

Array–pointer interchangeability

The subscript notation `x[i]` (where `x` designates a pointer) is syntactic sugar for `*(x+i)`.^[41] Taking advantage of the compiler's knowledge of the pointer type, the address that `x + i` points to is not the base address (pointed to by `x`) incremented by `i` bytes, but rather is defined to be the base address incremented by `i` multiplied by the size of an element that `x` points to. Thus, `x[i]` designates the `i+1`th element of the array.

Furthermore, in most expression contexts (a notable exception is as operand of `sizeof`), an expression of array type is automatically converted to a pointer to the array's first element. This implies that an array is never copied as a whole when named as an argument to a function, but rather only the address of its first element is passed. Therefore, although function calls in C use pass-by-value semantics, arrays are in effect passed by reference.

The total size of an array `x` can be determined by applying `sizeof` to an expression of array type. The size of an element can be determined by applying the operator `sizeof` to any dereferenced element of an array `A`, as in `n = sizeof A[0]`. Thus, the number of elements in a declared array `A` can be determined as `sizeof A / sizeof A[0]`. Note, that if only a pointer to the first element is available as it is often the case in C code because of the automatic conversion described above, the information about the full type of the array and its length are lost.

Memory management

One of the most important functions of a programming language is to provide facilities for managing memory and the objects that are stored in memory. C provides three principal ways to allocate memory for objects:^[36]

- Static memory allocation: space for the object is provided in the binary at compile-time; these objects have an extent (or lifetime) as long as the binary which contains them is loaded into memory.
- Automatic memory allocation: temporary objects can be stored on the stack, and this space is automatically freed and reusable after the block in which they are declared is exited.
- Dynamic memory allocation: blocks of memory of arbitrary size can be requested at run-time using library functions such as `malloc` from a region of memory called the heap; these blocks persist until subsequently freed for reuse by calling the library function `realloc` or `free`

These three approaches are appropriate in different situations and have various trade-offs. For example, static memory allocation has little allocation overhead, automatic allocation may involve slightly more overhead, and dynamic memory allocation can potentially have a great deal of overhead for both allocation and deallocation. The persistent nature of static objects is useful for maintaining state information across function calls, automatic allocation is easy to use but stack space is typically much more limited and transient than either static memory or heap space, and dynamic memory allocation allows convenient allocation of objects whose size is known only at run-time. Most C programs make extensive use of all three.

Where possible, automatic or static allocation is usually simplest because the storage is managed by the compiler, freeing the programmer of the potentially error-prone chore of manually allocating and releasing storage. However, many data structures can change in size at runtime, and since static allocations (and automatic allocations before C99) must have a fixed size at compile-time, there are many situations in which dynamic allocation is necessary.^[36] Prior to the C99 standard, variable-sized arrays were a common example of this. (See the article on `malloc` for an example of dynamically allocated arrays.) Unlike automatic allocation, which can fail at run time with uncontrolled consequences, the dynamic allocation functions return an indication (in the form of a null pointer value) when the required storage cannot be allocated. (Static allocation that is too large is usually detected by the linker or loader, before the program can even begin execution.)

Unless otherwise specified, static objects contain zero or null pointer values upon program startup. Automatically and dynamically allocated objects are initialized only if an initial value is explicitly specified; otherwise they initially have indeterminate values (typically, whatever bit pattern happens to be present in the storage, which might not even represent a valid value for that type). If the program attempts to access an uninitialized value, the results are undefined. Many modern compilers try to detect and warn about this problem, but both false positives and false negatives can occur.

Heap memory allocation has to be synchronized with its actual usage in any program to be reused as much as possible. For example, if the only pointer to a heap memory allocation goes out of scope or has its value overwritten before it is deallocated explicitly, then that memory cannot be recovered for later reuse and is essentially lost to the program, a phenomenon known as a memory leak. Conversely, it is possible for memory to be freed, but is referenced subsequently, leading to unpredictable results. Typically, the failure symptoms appear in a portion of the program unrelated to the code that causes the error, making it difficult to diagnose the failure. Such issues are ameliorated in languages with automatic garbage collection.

Libraries

The C programming language uses libraries as its primary method of extension. In C, a library is a set of functions contained within a single "archive" file. Each library typically has a header file, which contains the prototypes of the functions contained within the library that may be used by a program, and declarations of special data types and macro symbols used with these functions. In order for a program to use a library, it must include the library's header file, and the library must be linked with the program, which in many cases requires compiler flags (e.g., `-lm`, shorthand for "link the math library").^[36]

The most common C library is the C standard library, which is specified by the ISO and ANSI C

standards and comes with every C implementation (implementations which target limited environments such as embedded systems may provide only a subset of the standard library). This library supports stream input and output, memory allocation, mathematics, character strings, and time values. Several separate standard headers (for example, `stdio.h`) specify the interfaces for these and other standard library facilities.

Another common set of C library functions are those used by applications specifically targeted for Unix and Unix-like systems, especially functions which provide an interface to the kernel. These functions are detailed in various standards such as POSIX and the Single UNIX Specification.

Since many programs have been written in C, there are a wide variety of other libraries available. Libraries are often written in C because C compilers generate efficient object code; programmers then create interfaces to the library so that the routines can be used from higher-level languages like Java, Perl, and Python.^[36]

File handling and streams

File input and output (I/O) is not part of the C language itself but instead is handled by libraries (such as the C standard library) and their associated header files (e.g. `stdio.h`). File handling is generally implemented through high-level I/O which works through streams. A stream is from this perspective a data flow that is independent of devices, while a file is a concrete device. The high-level I/O is done through the association of a stream to a file. In the C standard library, a buffer (a memory area or queue) is temporarily used to store data before it is sent to the final destination. This reduces the time spent waiting for slower devices, for example a hard drive or solid state drive. Low-level I/O functions are not part of the standard C library but are generally part of "bare metal" programming (programming that's independent of any operating system such as most embedded programming). With few exceptions, implementations include low-level I/O.

Language tools

A number of tools have been developed to help C programmers find and fix statements with undefined behavior or possibly erroneous expressions, with greater rigor than that provided by the compiler. The tool lint was the first such, leading to many others.

Automated source code checking and auditing are beneficial in any language, and for C many such tools exist, such as Lint. A common practice is to use Lint to detect questionable code when a program is first written. Once a program passes Lint, it is then compiled using the C compiler. Also, many compilers can optionally warn about syntactically valid constructs that are likely to actually be errors. MISRA C is a proprietary set of guidelines to avoid such questionable code, developed for embedded systems.^[42]

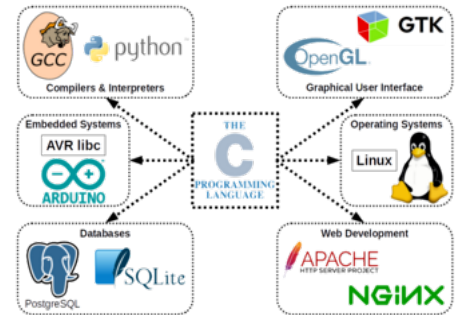
There are also compilers, libraries, and operating system level mechanisms for performing actions that are not a standard part of C, such as bounds checking for arrays, detection of buffer overflow, serialization, dynamic memory tracking, and automatic garbage collection.

Tools such as Purify or Valgrind and linking with libraries containing special versions of the memory allocation functions can help uncover runtime errors in memory usage.^{[43][44]}

Uses

Rationale for use in systems programming

C is widely used for systems programming in implementing operating systems and embedded system applications.^[45] This is for several reasons:



The C programming language

- The code generated after compilation does not demand many system features, and can be invoked from some boot code in a straightforward manner – it is simple to execute.
- The C language statements and expressions typically map well on to sequences of instructions for the target processor, and consequently there is a low run-time demand on system resources – it is fast to execute.
- With its rich set of operators, the C language can utilise many of the features of target CPUs. Where a particular CPU has more esoteric instructions, a language variant can be constructed with perhaps intrinsic functions to exploit those instructions – it can use practically all the target CPU's features.
- The language makes it easy to overlay structures onto blocks of binary data, allowing the data to be comprehended, navigated and modified – it can write data structures, even file systems.
- The language supports a rich set of operators, including bit manipulation, for integer arithmetic and logic, and perhaps different sizes of floating point numbers – it can process appropriately-structured data effectively.
- C is a fairly small language, with only a handful of statements, and without too many features that generate extensive target code – it is comprehensible.
- C has direct control over memory allocation and deallocation, which gives reasonable efficiency and predictable timing to memory-handling operations, without any concerns for sporadic stop-the-world garbage collection events – it has predictable performance.
- Platform hardware can be accessed with pointers and type punning, so system-specific features (e.g. Control/Status Registers, I/O registers) can be configured and used with code written in C – it interacts well with the platform it is running on.
- Depending on the linker and environment, C code can also call libraries written in assembly language, and may be called from assembly language – it interoperates well with other lower-level code.
- C and its calling conventions and linker structures are commonly used in conjunction with other high-level languages, with calls both to C and from C supported – it interoperates well with other high-level code.
- C has a very mature and broad ecosystem, including libraries, frameworks, open source compilers, debuggers and utilities, and is the de facto standard. It is likely the drivers already exist in C, or that there is a similar CPU architecture as a back-end of a C compiler, so there is reduced incentive to choose another language.

Once used for web development

Historically, C was sometimes used for web development using the Common Gateway Interface

(CGI) as a "gateway" for information between the web application, the server, and the browser.^[46] C may have been chosen over interpreted languages because of its speed, stability, and near-universal availability.^[47] It is no longer common practice for web development to be done in C,^[48] and many other web development tools exist.

Some other languages are themselves written in C

A consequence of C's wide availability and efficiency is that compilers, libraries and interpreters of other programming languages are often implemented in C.^[49] For example, the reference implementations of Python,^[50] Perl,^[51] Ruby,^[52] and PHP^[53] are written in C.

Used for computationally-intensive libraries

C enables programmers to create efficient implementations of algorithms and data structures, because the layer of abstraction from hardware is thin, and its overhead is low, an important criterion for computationally intensive programs. For example, the GNU Multiple Precision Arithmetic Library, the GNU Scientific Library, Mathematica, and MATLAB are completely or partially written in C. Many languages support calling library functions in C, for example, the Python-based framework NumPy uses C for the high-performance and hardware-interacting aspects.

C as an intermediate language

C is sometimes used as an intermediate language by implementations of other languages. This approach may be used for portability or convenience; by using C as an intermediate language, additional machine-specific code generators are not necessary. C has some features, such as line-number preprocessor directives and optional superfluous commas at the end of initializer lists, that support compilation of generated code. However, some of C's shortcomings have prompted the development of other C-based languages specifically designed for use as intermediate languages, such as C--. Also, contemporary major compilers GCC and LLVM both feature an intermediate representation that is not C, and those compilers support front ends for many languages including C.

End-user applications

C has also been widely used to implement end-user applications. However, such applications can also be written in newer, higher-level languages.

Limitations

the power of assembly language and the convenience of ... assembly language

—Dennis Ritchie^[54]

While C has been popular, influential and hugely successful, it has drawbacks, including:

- The standard dynamic memory handling with `malloc` and `free` is error prone. Bugs include: Memory leaks when memory is allocated but not freed; and access to previously freed memory.
- The use of pointers and the direct manipulation of memory means corruption of memory is possible, perhaps due to programmer error, or insufficient checking of bad data.
- There is some type checking, but it does not apply to areas like variadic functions, and the type checking can be trivially or inadvertently circumvented. It is weakly typed.
- Since the code generated by the compiler contains few checks itself, there is a burden on the programmer to consider all possible outcomes, to protect against buffer overruns, array bounds checking, stack overflows, memory exhaustion, and consider race conditions, thread isolation, etc.
- The use of pointers and the run-time manipulation of these means there may be two ways to access the same data (aliasing), which is not determinable at compile time. This means that some optimisations that may be available to other languages are not possible in C. FORTRAN is considered faster.
- Some of the standard library functions, e.g. `scanf` or `strncat`, can lead to buffer overruns.
- There is limited standardisation in support for low-level variants in generated code, for example: different function calling conventions and ABI; different structure packing conventions; different byte ordering within larger integers (including endianness). In many language implementations, some of these options may be handled with the preprocessor directive `#pragma`,^{[55][56]} and some with additional keywords e.g. use `__cdecl` calling convention. But the directive and options are not consistently supported.^[57]
- String handling using the standard library is code-intensive, with explicit memory management required.
- The language does not directly support object orientation, introspection, run-time expression evaluation, generics, etc.
- There are few guards against inappropriate use of language features, which may lead to unmaintainable code. In particular, the C preprocessor can hide troubling effects such as double evaluation and worse.^[58] This facility for tricky code has been celebrated with competitions such as the *International Obfuscated C Code Contest* and the *Underhanded C Contest*.
- C lacks standard support for exception handling and only offers return codes for error checking. The `setjmp` and `longjmp` standard library functions have been used^[59] to implement a try-catch mechanism via macros.

For some purposes, restricted styles of C have been adopted, e.g. MISRA C or CERT C, in an attempt to reduce the opportunity for bugs. Databases such as CWE attempt to count the ways C etc. has vulnerabilities, along with recommendations for mitigation.

There are tools that can mitigate against some of the drawbacks. Contemporary C compilers include checks which may generate warnings to help identify many potential bugs.

Some of these drawbacks have prompted the construction of other languages.

Related languages

C has both directly and indirectly influenced many later languages such as C++ and Java.^[61] The most pervasive influence has been syntactical; all of the languages mentioned combine the

statement and (more or less recognizably) expression syntax of C with type systems, data models or large-scale program structures that differ from those of C, sometimes radically.

Several C or near-C interpreters exist, including Ch and CINT, which can also be used for scripting.

When object-oriented programming languages became popular, C++ and Objective-C were two different extensions of C that provided object-oriented capabilities. Both languages were originally implemented as source-to-source compilers; source code was translated into C, and then compiled with a C compiler.^[62]

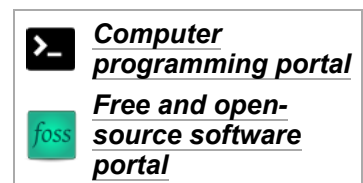
The C++ programming language (originally named "C with Classes") was devised by Bjarne Stroustrup as an approach to providing object-oriented functionality with a C-like syntax.^[63] C++ adds greater typing strength, scoping, and other tools useful in object-oriented programming, and permits generic programming via templates. Nearly a superset of C, C++ now supports most of C, with a few exceptions.

Objective-C was originally a very "thin" layer on top of C, and remains a strict superset of C that permits object-oriented programming using a hybrid dynamic/static typing paradigm. Objective-C derives its syntax from both C and Smalltalk: syntax that involves preprocessing, expressions, function declarations, and function calls is inherited from C, while the syntax for object-oriented features was originally taken from Smalltalk.

In addition to C++ and Objective-C, Ch, Cilk, and Unified Parallel C are nearly supersets of C.

See also

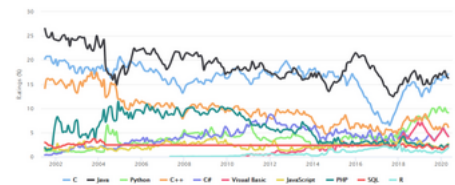
- Compatibility of C and C++
- Comparison of Pascal and C
- Comparison of programming languages
- International Obfuscated C Code Contest
- List of C-based programming languages
- List of C compilers



Notes

- The original example code will compile on most modern compilers that are not in strict standard compliance mode, but it does not fully conform to the requirements of either C89 or C99. In fact, C99 requires that a diagnostic message be produced.
- The main function actually has two arguments, `int argc` and `char *argv[]`, respectively, which can be used to handle command-line arguments. The ISO C standard (section 5.1.2.2.1) requires both forms of main to be supported, which is special treatment not afforded to any other function.

References



The TIOBE index graph, showing a comparison of the popularity of various programming languages^[60]

1. Prinz, Peter; Crawford, Tony (December 16, 2005). *C in a Nutshell* (<https://books.google.com/books?id=4Mfe4sAMFUyC>). O'Reilly Media, Inc. p. 3. ISBN 9780596550714.
2. Ritchie (1993): "Thompson had made a brief attempt to produce a system coded in an early version of C—before structures—in 1972, but gave up the effort."
3. ISO/IEC JTC1/SC22/WG14 (April 5, 2023). "C - Project status and milestones" (<https://www.open-std.org/jtc1/sc22/wg14/www/projects>). Retrieved August 9, 2023.
4. Ritchie (1993): "The scheme of type composition adopted by C owes considerable debt to Algol 68, although it did not, perhaps, emerge in a form that Algol's adherents would approve of."
5. "Verilog HDL (and C)" (https://web.archive.org/web/20131106064022/http://cs.anu.edu.au/courses/ENGN3213/lectures/lecture6_VERILOG_2010.pdf) (PDF). The Research School of Computer Science at the Australian National University. June 3, 2010. Archived from the original (http://cs.anu.edu.au/courses/ENGN3213/lectures/lecture6_VERILOG_2010.pdf) (PDF) on November 6, 2013. Retrieved August 19, 2013. "1980s: ; Verilog first introduced ; Verilog inspired by the C programming language"
6. "The name is based on, and pronounced like the letter C in the English alphabet" (<https://eng.ichacha.net/pronounce/the%20c%20programming%20language.html>). *the c programming language sound*. English Chinese Dictionary. Archived (<https://web.archive.org/web/20221117151137/https://eng.ichacha.net/pronounce/the%20c%20programming%20language.html>) from the original on November 17, 2022. Retrieved November 17, 2022.
7. "C Language Drops to Lowest Popularity Rating" (<https://www.developer.com/news/c-language-drops-to-lowest-popularity-rating/>). *Developer.com*. August 9, 2016. Archived (<https://web.archive.org/web/20220822225609/https://www.developer.com/news/c-language-drops-to-lowest-popularity-rating/>) from the original on August 22, 2022. Retrieved August 1, 2022.
8. Ritchie (1993)
9. "Programming Language Popularity" (<https://web.archive.org/web/20090116080326/http://www.langpop.com/>). 2009. Archived from the original (<http://www.langpop.com/>) on January 16, 2009. Retrieved January 16, 2009.
10. "TIOBE Programming Community Index" (<https://web.archive.org/web/20090504181627/http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>). 2009. Archived from the original (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>) on May 4, 2009. Retrieved May 6, 2009.
11. Ward, Terry A. (August 1983). "Annotated C / A Bibliography of the C Language" (https://archive.org/stream/byte-magazine-1983-08/1983_08_BYTE_08-08_The_C_Language#page/n267/mode/2up). *Byte*. p. 268. Retrieved January 31, 2015.
12. Prinz, Peter; Crawford, Tony (December 16, 2005). *C in a Nutshell* (<https://books.google.com/books?id=4Mfe4sAMFUyC>). O'Reilly Media, Inc. p. 3. ISBN 9780596550714.
13. "History of C" (<http://en.cppreference.com/w/c/language/history>). *en.cppreference.com*. Archived (<https://web.archive.org/web/20180529130541/http://en.cppreference.com/w/c/language/history>) from the original on May 29, 2018. Retrieved May 28, 2018.
14. "TIOBE Index for October 2021" (<https://www.tiobe.com/tiobe-index/>). Archived (<https://web.archive.org/web/20180225101948/https://www.tiobe.com/tiobe-index/>) from the original on February 25, 2018. Retrieved October 7, 2021.
15. Ritchie, Dennis. "BCPL to B to C" (<https://www.lysator.liu.se/c/dmr-on-histories.html>). Archived (<https://web.archive.org/web/20191212221532/http://www.lysator.liu.se/c/dmr-on-histories.html>) from the original on December 12, 2019. Retrieved September 10, 2019.

16. Jensen, Richard (December 9, 2020). "A damn stupid thing to do"—the origins of C" (<https://arstechnica.com/features/2020/12/a-damn-stupid-thing-to-do-the-origins-of-c/>). *Ars Technica*. Archived (<https://web.archive.org/web/20220328143845/https://arstechnica.com/features/2020/12/a-damn-stupid-thing-to-do-the-origins-of-c/>) from the original on March 28, 2022. Retrieved March 28, 2022.
17. Johnson, S. C.; Ritchie, D. M. (1978). "Portability of C Programs and the UNIX System". *Bell System Tech. J.* **57** (6): 2021–2048. CiteSeerX 10.1.1.138.35 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.35>). doi:10.1002/j.1538-7305.1978.tb02141.x (<https://doi.org/10.1002%2Fj.1538-7305.1978.tb02141.x>). S2CID 17510065 (<https://api.semanticscholar.org/CorpusID:17510065>). (Note: The PDF is an OCR scan of the original, and contains a rendering of "IBM 370" as "IBM 310".)
18. McIlroy, M. D. (1987). *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986* (<http://www.cs.dartmouth.edu/~doug/reader.pdf>) (PDF) (Technical report). CSTR. Bell Labs. p. 10. 139. Archived (<https://web.archive.org/web/20171111151817/http://www.cs.dartmouth.edu/~doug/reader.pdf>) (PDF) from the original on November 11, 2017. Retrieved February 1, 2015.
19. Kernighan, Brian W.; Ritchie, Dennis M. (February 1978). *The C Programming Language* (1st ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 978-0-13-110163-0.
20. "C manual pages". *FreeBSD Miscellaneous Information Manual* (<https://nxbmnpng.lemoda.net/7/c78>) (FreeBSD 13.0 ed.). May 30, 2011. Archived (<https://web.archive.org/web/20210121024455/https://nxbmnpng.lemoda.net/7/c78>) from the original on January 21, 2021. Retrieved January 15, 2021. [1] (<https://www.freebsd.org/cgi/man.cgi?query=c78&apropos=0&sektion=0&manpath=FreeBSD+9-current&arch=default&format=html>) Archived (<https://web.archive.org/web/20210121033654/https://www.freebsd.org/cgi/man.cgi?query=c78&apropos=0&sektion=0&manpath=FreeBSD+9-current&arch=default&format=html>) January 21, 2021, at the Wayback Machine
21. Kernighan, Brian W.; Ritchie, Dennis M. (March 1988). *The C Programming Language* (2nd ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 978-0-13-110362-7.
22. Stroustrup, Bjarne (2002). Sibling rivalry: C and C++ (http://stroustrup.com/sibling_rivalry.pdf) (PDF) (Report). AT&T Labs. Archived (https://web.archive.org/web/20140824072719/http://www.stroustrup.com/sibling_rivalry.pdf) (PDF) from the original on August 24, 2014. Retrieved April 14, 2014.
23. *C Integrity* (<https://www.iso.org/standard/23909.html>). International Organization for Standardization. March 30, 1995. Archived (<https://web.archive.org/web/20180725033429/http://www.iso.org/standard/23909.html>) from the original on July 25, 2018. Retrieved July 24, 2018.
24. "JTC1/SC22/WG14 – C" (<http://www.open-std.org/jtc1/sc22/wg14/>). *Home page*. ISO/IEC. Archived (<https://web.archive.org/web/20180212100115/http://www.open-std.org/JTC1/SC22/WG14/>) from the original on February 12, 2018. Retrieved June 2, 2011.
25. Andrew Binstock (October 12, 2011). "Interview with Herb Sutter" (<http://www.drdobbs.com/cpp/interview-with-herb-sutter/231900562>). *Dr. Dobbs*. Archived (<https://web.archive.org/web/20130802070446/http://www.drdobbs.com/cpp/interview-with-herb-sutter/231900562>) from the original on August 2, 2013. Retrieved September 7, 2013.
26. "WG14-N3132 : Revised C23 Schedule" (<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3132.pdf>) (PDF). *open-std.org*. June 4, 2023. Archived (<https://web.archive.org/web/20230609204739/https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3132.pdf>) (PDF) from the original on June 9, 2023.

27. "TR 18037: Embedded C" (<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf>) (PDF). ISO / IEC. Archived (<https://web.archive.org/web/20210225224616/http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf>) (PDF) from the original on February 25, 2021. Retrieved July 26, 2011.
28. Harbison, Samuel P.; Steele, Guy L. (2002). *C: A Reference Manual* (5th ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 978-0-13-089592-9. Contains a BNF grammar for C.
29. Kernighan & Ritchie (1988), p. 192.
30. Kernighan & Ritchie (1978), p. 3.
31. "ISO/IEC 9899:201x (ISO C11) Committee Draft" (<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf>) (PDF). Archived (<https://web.archive.org/web/20171222215122/http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf>) (PDF) from the original on December 22, 2017. Retrieved September 16, 2011.
32. Kernighan & Ritchie (1988), pp. 192, 259.
33. "10 Common Programming Mistakes in C++" (<http://www.cs.ucr.edu/~nxiao/cs10/errors.htm>). Cs.ucr.edu. Archived (<https://web.archive.org/web/20081021080953/http://www.cs.ucr.edu/~nxiao/cs10/errors.htm>) from the original on October 21, 2008. Retrieved June 26, 2009.
34. Schultz, Thomas (2004). *C and the 8051* (<https://books.google.com/books?id=rI0c8kWBxooC&pg=PT47>) (3rd ed.). Otsego, MI: PageFree Publishing Inc. p. 20. ISBN 978-1-58961-237-2. Retrieved February 10, 2012.
35. Kernighan & Ritchie (1978), p. 6.
36. Klemens, Ben (2013). *21st Century C*. O'Reilly Media. ISBN 978-1-4493-2714-9.
37. Feuer, Alan R.; Gehani, Narain H. (March 1982). "Comparison of the Programming Languages C and Pascal". *ACM Computing Surveys*. **14** (1): 73–92. doi:10.1145/356869.356872 (<https://doi.org/10.1145%2F356869.356872>). S2CID 3136859 (<https://api.semanticscholar.org/CorpusID:3136859>).
38. Kernighan & Ritchie (1988), p. 122.
39. For example, gcc provides `_FORTIFY_SOURCE`. "Security Features: Compile Time Buffer Checks (FORTIFY_SOURCE)" (<https://fedoraproject.org/wiki/Security/Features>). fedoraproject.org. Archived (<https://web.archive.org/web/20070107153447/http://fedoraproject.org/wiki/Security/Features>) from the original on January 7, 2007. Retrieved August 5, 2012.
40. เอี่ยมสิริวงศ์, โภาศ (2016). *Programming with C*. Bangkok, Thailand: SE-EDUCATION PUBLIC COMPANY LIMITED. pp. 225–230. ISBN 978-616-08-2740-4.
41. Raymond, Eric S. (October 11, 1996). *The New Hacker's Dictionary* (https://books.google.com/books?id=g80P_4v4QbIC&pg=PA432) (3rd ed.). MIT Press. p. 432. ISBN 978-0-262-68092-9. Retrieved August 5, 2012.
42. "Man Page for lint (freebsd Section 1)" (<http://www.unix.com/man-page/FreeBSD/1/lint>). *unix.com*. May 24, 2001. Retrieved July 15, 2014.
43. "CS107 Valgrind Memcheck" (<https://web.stanford.edu/class/archive/cs/cs107/cs107.1236/resources/valgrind.html>). *web.stanford.edu*. Retrieved June 23, 2023.
44. Hastings, Reed; Joyce, Bob. "Purify: Fast Detection of Memory Leaks and Access Errors" (<https://web.stanford.edu/class/cs343/resources/purify.pdf>) (PDF). *Pure Software Inc.*: 9.
45. Dale, Nell B.; Weems, Chip (2014). *Programming and problem solving with C++* (6th ed.). Burlington, MA: Jones & Bartlett Learning. ISBN 978-1449694289. OCLC 894992484 (<https://www.worldcat.org/oclc/894992484>).
46. *Dr. Dobb's Sourcebook*. U.S.: Miller Freeman, Inc. November–December 1995.

47. "Using C for CGI Programming" (<http://www.linuxjournal.com/article/6863>). *linuxjournal.com*. March 1, 2005. Archived (<https://web.archive.org/web/20100213075858/http://www.linuxjournal.com/article/6863>) from the original on February 13, 2010. Retrieved January 4, 2010.
48. Perkins, Luc (September 17, 2013). "Web development in C: crazy? Or crazy like a fox?" (<http://medium.com/@lucperkins/web-development-in-c-crazy-or-crazy-like-a-fox-ff723209f8f5>). *Medium*. Archived (<https://web.archive.org/web/20141004135317/https://medium.com/@lucperkins/web-development-in-c-crazy-or-crazy-like-a-fox-ff723209f8f5>) from the original on October 4, 2014. Retrieved April 8, 2022.
49. "C - the mother of all languages" (<https://ict.iitk.ac.in/c-the-mother-of-all-languages/>). *ICT Academy at IITK*. November 13, 2018. Archived (<https://web.archive.org/web/20210531161841/https://ict.iitk.ac.in/c-the-mother-of-all-languages/>) from the original on May 31, 2021. Retrieved October 11, 2022.
50. "1. Extending Python with C or C++ — Python 3.10.7 documentation" (<https://docs.python.org/3/extending/extending.html>). *docs.python.org*. Archived (<https://web.archive.org/web/20121105232707/https://docs.python.org/3/extending/extending.html>) from the original on November 5, 2012. Retrieved October 11, 2022.
51. "An overview of the Perl 5 engine | Opensource.com" (<https://opensource.com/article/18/1/perl-5-engine>). *opensource.com*. Archived (<https://web.archive.org/web/20220526105419/https://opensource.com/article/18/1/perl-5-engine>) from the original on May 26, 2022. Retrieved October 11, 2022.
52. "To Ruby From C and C++" (<https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-c-and-cpp/>). *www.ruby-lang.org*. Archived (<https://web.archive.org/web/20130812003928/https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-c-and-cpp/>) from the original on August 12, 2013. Retrieved October 11, 2022.
53. "What is PHP? How to Write Your First PHP Program" (<https://www.freecodecamp.org/news/what-is-php-write-your-first-php-program/>). *freeCodeCamp.org*. August 3, 2022. Archived (<https://web.archive.org/web/20220804050401/https://www.freecodecamp.org/news/what-is-php-write-your-first-php-program/>) from the original on August 4, 2022. Retrieved October 11, 2022.
54. Metz, Cade. "Dennis Ritchie: The Shoulders Steve Jobs Stood On" (<https://www.wired.com/2011/10/thedennisritchieeffect/>). *Wired*. Archived (<https://web.archive.org/web/20220412005125/http://www.wired.com/2011/10/thedennisritchieeffect/>) from the original on April 12, 2022. Retrieved April 19, 2022.
55. corob-msft (March 31, 2022). "Pragma directives and the `__pragma` and `_Pragma` keywords" (<https://learn.microsoft.com/en-us/cpp/preprocessor/pragma-directives-and-the-pragma-keyword>). *learn.microsoft.com*. Archived (<https://web.archive.org/web/20220924075131/https://learn.microsoft.com/en-us/cpp/preprocessor/pragma-directives-and-the-pragma-keyword>) from the original on September 24, 2022. Retrieved September 24, 2022.
56. "Pragmas (The C Preprocessor)" (<https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html>). *gcc.gnu.org*. Archived (<https://web.archive.org/web/20020617041757/https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html>) from the original on June 17, 2002. Retrieved September 24, 2022.
57. "Pragmas" (<https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/pragmas.html>). *Intel*. Archived (<https://web.archive.org/web/20220410113529/https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/pragmas.html>) from the original on April 10, 2022. Retrieved April 10, 2022.
58. "In praise of the C preprocessor" (<https://apenwarr.ca/log/20070813>). *apenwarr.ca*. Retrieved July 9, 2023.

59. Roberts, Eric S. (March 21, 1989). "Implementing Exceptions in C" (http://bitsavers.informatik.uni-stuttgart.de/pdf/dec/tech_reports/SRC-RR-40.pdf) (PDF). DEC Systems Research Center. SRC-RR-40. Archived (https://web.archive.org/web/20170115152453/http://bitsavers.informatik.uni-stuttgart.de/pdf/dec/tech_reports/SRC-RR-40.pdf) (PDF) from the original on January 15, 2017. Retrieved January 4, 2022.
60. McMillan, Robert (August 1, 2013). "Is Java Losing Its Mojo?" (<https://www.wired.com/2013/01/java-no-longer-a-favorite/>). *Wired*. Archived (<https://web.archive.org/web/20170215115409/http://www.wired.com/2013/01/java-no-longer-a-favorite/>) from the original on February 15, 2017. Retrieved March 5, 2017.
61. O'Regan, Gerard (September 24, 2015). *Pillars of computing : a compendium of select, pivotal technology firms*. Springer. ISBN 978-3319214641. OCLC 922324121 (<https://www.worldcat.org/oclc/922324121>).
62. Rauchwerger, Lawrence (2004). *Languages and compilers for parallel computing : 16th international workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003 : revised papers*. Springer. ISBN 978-3540246442. OCLC 57965544 (<https://www.worldcat.org/oclc/57965544>).
63. Stroustrup, Bjarne (1993). "A History of C++: 1979–1991" (<http://www.stroustrup.com/hopl2.pdf>) (PDF). Archived (<https://web.archive.org/web/20190202050609/http://www.stroustrup.com/hopl2.pdf>) (PDF) from the original on February 2, 2019. Retrieved June 9, 2011.

Sources

- Ritchie, Dennis M. (March 1993). "The Development of the C Language" (<https://doi.org/10.1145%2F155360.155580>). *ACM SIGPLAN Notices*. ACM. **28** (3): 201–208. doi:10.1145/155360.155580 (<https://doi.org/10.1145%2F155360.155580>).
 - By courtesy of the author, also at Ritchie, Dennis M. "Chistory" (<https://www.bell-labs.com/usr/dmr/www/chist.html>). *www.bell-labs.com*. Retrieved March 29, 2022.
- Ritchie, Dennis M. (1993). "The Development of the C Language" (<http://www.bell-labs.com/usr/dmr/www/chist.html>). *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM. pp. 201–208. doi:10.1145/154766.155580 (<https://doi.org/10.1145%2F154766.155580>). ISBN 0-89791-570-4. Retrieved November 4, 2014.
- Kernighan, Brian W.; Ritchie, Dennis M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall. ISBN 0-13-110362-8.

Further reading

- Plauger, P.J. (1992). *The Standard C Library* (1 ed.). Prentice Hall. ISBN 978-0131315099. (*source*) (https://github.com/wuzhouhui/c_standard_lib)
- Banahan, M.; Brady, D.; Doran, M. (1991). *The C Book: Featuring the ANSI C Standard* (2 ed.). Addison-Wesley. ISBN 978-0201544336. (*free*) (<https://github.com/wardvanwanrooij/thecbook>)
- Harbison, Samuel; Steele, Guy Jr. (2002). *C: A Reference Manual* (5 ed.). Pearson. ISBN 978-0130895929. (*archive*) (<https://archive.org/details/creferencemanual00harb>)
- King, K.N. (2008). *C Programming: A Modern Approach* (2 ed.). W. W. Norton. ISBN 978-0393979503. (*archive*) (<https://archive.org/details/cprogrammingmode0000king>)
- Griffiths, David; Griffiths, Dawn (2012). *Head First C* (1 ed.). O'Reilly. ISBN 978-1449399917.
- Perry, Greg; Miller, Dean (2013). *C Programming: Absolute Beginner's Guide* (3 ed.). Que.

ISBN 978-0789751980.

- Deitel, Paul; Deitel, Harvey (2015). *C: How to Program* (8 ed.). Pearson. ISBN 978-0133976892.
- Gustedt, Jens (2019). *Modern C* (2 ed.). Manning. ISBN 978-1617295812. *(free)* (<https://gustedt.github.io/pages/inria.fr/modern-c/>)

External links

- ISO C Working Group official website (<https://www.open-std.org/jtc1/sc22/wg14/>)
 - ISO/IEC 9899 (<https://www.open-std.org/JTC1/SC22/WG14/www/standards>), publicly available official C documents, including the C99 Rationale
 - "C99 with Technical corrigenda TC1, TC2, and TC3 included" (<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>) (PDF). Archived (<https://web.archive.org/web/20071025205438/http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>) (PDF) from the original on October 25, 2007. (3.61 MB)
 - comp.lang.c Frequently Asked Questions (<https://c-faq.com/>)
 - A History of C (<http://csapp.cs.cmu.edu/3e/docs/chistory.html>), by Dennis Ritchie
 - C Library Reference and Examples (<https://en.cppreference.com/w/c>)
-

Retrieved from "[https://en.wikipedia.org/w/index.php?title=C_\(programming_language\)&oldid=1179312467](https://en.wikipedia.org/w/index.php?title=C_(programming_language)&oldid=1179312467)"

▪