A programming language called C

Brian W. Kernighan

The C programming language is claimed to be compact, efficient, and expressive, to the point of supplanting assembly language on Unix

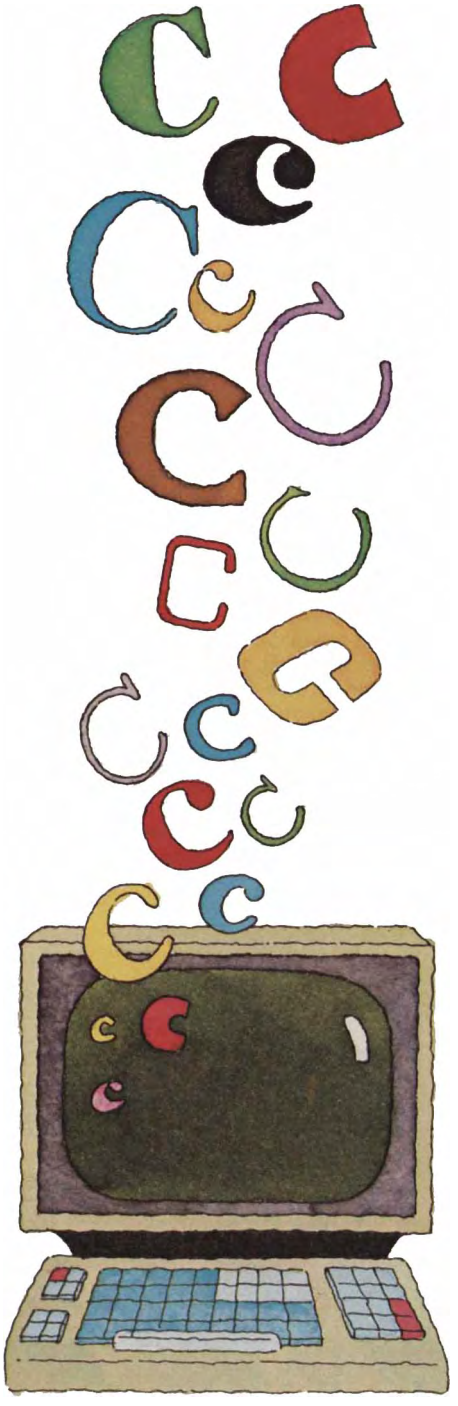
C is a general-purpose programming language developed by Dennis Ritchie at Bell Laboratories in 1972. Since then, it has become a major language not only at Bell Labs but also throughout the world. C was originally developed for use with the Unix [a trademark of Bell Labs] operating system, which is largely written in C, so part of the success of the language is due to the acceptance of Unix. C, however, has spread far beyond Unix systems in the past few years, and a booming compiler industry has sprung up around it.

C was originally designed for "systems programming," that is, for writing programs like compilers, operating systems, and text editors. It has proven satisfactory for other applications as well, including data base systems, telephone switching systems, numerical analysis, engineering programs, and a great deal of text-processing software.

To help set C in perspective, it can be compared to other widely known languages such as Basic, Fortran, Pascal, and Ada. Each of these languages has its own area of special application but can be made to cope with problems outside this domain, although sometimes with difficulty.

At the risk of offending proponents of other languages, I would summarize the distinctions in this way: Basic is meant for small, quickly developed, interactive programs with an emphasis on numeric processing. It is well suited for 10- to 20-line programs that compute a set of numbers, but it does not scale well to larger programs. Basic is very easy to learn, a major asset. It is popular for microcomputers because it can be implemented in a very small memory.

Fortran is meant for large scientific and engineering calculations. Much Fortran-compatible software is available, primarily for numerical work. Fortran, like Basic, provides the programmer with little help in making programs easy to read. Neither language provides much support for complicated data structures.



Pascal was designed as a teaching language, especially for computer science. It is small and clean, although quite limited in many ways. It provides reasonable tools for readability and complicated data structures. Ada is derived from Pascal, but it is a far bigger language. Although Ada is intended for "embedded systems" that may be roughly translated into real-time computations, it is a general-purpose language that should have broad applications. Ada has not seen enough real use in any area to be evaluated fairly.

History and evolution

C has its roots in the language BCPL, which is a "typeless" language that operates only on a single data type, the machine word. As such, BCPL is an excellent match to the hardware of word-oriented machines such as the PDP-10. In 1970, Ken Thompson designed a stripped-down version of BCPL for use with the first Unix system on the PDP-7. This stripped-down version is called B, and like BCPL is typeless.

With the advent of the PDP-11, upon which the next version of Unix was written, it became clear that a typeless language did not match the hardware's capabilities. The PDP-11 provided several fundamental objects of different sizes—1-byte characters, 2-byte integers, and 4-byte floating point numbers. B provided no way to talk about these different sizes, let alone operators to manipulate them.

The C language was an attempt to deal with a variety of types, which it

0278-6648/83/0400-0026$1.00 © 1983

did by adding the notion of data type to the B language. In C, as in most languages, each object has a type as well as a value. The type determines what kinds of machine operations can be applied to the value and how much storage is occupied. For example, the following declarations for Fortran and C determine the operations and space requirements of the variables:

```
Fortran:          C:
  INTEGER I, J      int i, j;
  DOUBLE D          double d;
  REAL X            float x;
```

In the statement

```
  Fortran:        C:
D = X + I * J       d = x + i * j;
```

the compiler uses the type information to determine that integer multiplication is adequate for I * J, but the result has to be converted to floating point before it is added to X and then to double precision for assignment to D.

Although C was originally implemented for a PDP-11, it is not particularly tied to that machine, and around 1975 work began at Bell Labs on C compilers for other machines. In particular, a "portable compiler" was implemented, which made it relatively easy to modify the compiler to generate code for different machines.

As the Unix operating system spread, the technology of the portable compiler made it possible to move the operating system and its programs from one kind of hardware to another with little work. This meant that C became more widely available and used.

In the late 1970s, C became available from commercial sources, as well as from Bell Labs, for microprocessors like the Z80. This marked the beginning of C's commercial success. Since then, it has been implemented on many computers, from the smallest microprocessors to machines as large as the Cray-1; compilers are available from dozens of suppliers. The C language is sufficiently well standardized that with some care, it is possible to write C programs that will run without change on any machine that supports the standard language and the standard run-time environment.

A simple language

C began its life on a small machine, derived from a sequence of small languages. Its designer had a personal preference for simplicity and elegance rather than features. Furthermore, from the beginning C was meant for system programming applications, where efficiency matters. Accordingly, it is not too surprising that C is a good match to the capabilities of real machines. For example, C provides as its basic data type only objects that are directly supported by typical hardware: characters, integers (perhaps of several sizes), floating point numbers (in two sizes), and machine addresses (pointers). More complicated objects like arrays, structures, and so on, can be created, but C provides no operators for manipulating them as a unit: The user must write functions in order to do things like comparing strings, assigning one array to another, and so on.

Somewhat more unusual, C does not provide input and output operations as part of the language. This is not to say that C programs cannot do I/O, of course. I/O is done by functions defined by the user, not by built-in statements of the language. This is in contrast to Fortran's READ and WRITE and Basic's INPUT and PRINT, which are parts of the language.

There is also no storage management in C—like Pascal's "new" function—and no facilities for concurrent processing, as in Ada's rendezvous mechanism. These capabilities can be easily written in C, but are provided by libraries of functions, not as part of the language.

Language extension by functions is generally a good idea. There is no cost to the user who doesn't use a particular function. Furthermore, it allows the language to remain small, which makes it easier to learn and use and certainly easier to implement and support on a small machine. It is also easier to change properties that are defined by functions instead of being wired into the language.

On the negative side, however, function calls are notationally clumsier than direct operators. For example, compare Basic's string comparison

```
IF A$ = B$ THEN . . .
```

to the way it might be written in C

```
if (equal (a,b)) . . .
```

Function calls also involve more overhead than in-line code. Finally, it is harder to maintain standardization of features that are not part of the language.

In any case, the positive side outweighs the negative for C: The degree to which features are omitted from C is one of its distinguishing characteristics.

C programs

Let us illustrate the C language by writing a few small programs, the first of which merely prints **"hello world."** This is a good first program in any language, since it tests one's understanding of the compiler and the operating system before adding any problems from the language.

```
main ( )  /* prints 'hello world'
             message */
{
  printf("hello world\n");
}
```

Since the late 1970s, C has been implemented on many computers, from the smallest microprocessors to machines as large as the CRAY-1.

A C program is made up of a set of function definitions. Functions in C are like subroutines and functions in Fortran or procedures and functions in Pascal, except that C doesn't distinguish between subroutine and function. This program defines a function called **main.**

Every C program begins execution at the first statement of **main.** The first, and only, statement of this program calls another function **printf** that prints the string of characters between quotes. **printf** is a standard function for C programs, although it's not part of C, as explained above. It also has more capabilities than shown here, at least equal to WRITE in Fortran or Pascal. Execution of the program ends with the closing brace.

Let's write a bigger program. C doesn't provide an exponentiation operator like Fortran's ** (^ in Basic), but we can make a simple, integer-only, version by writing a function **power(m,n)** that produces the **n**-th power of the integer **m.**

```
main( ) /* test power function*/
{
  int i;     /* declare local
                integer variable */
  long power( ); /* declare type re-
                    turned by non-
                    std fcn */
  for (i = 0;  i <= 20; + +i)
    printf ("%d %ld\n",
      i, power(2,i));
}
```

```
long power(m, n)        /* raises m to
      n-th power, n>= 0 */
  int m, n;     /* declare
                   arguments  */
{
  int i;        /* declare local
                   variables */
  long p;

  p = 1;
  for (i = 1; i <= n; + +i)
    p = p * m;
  return p;
}
```

In C, as in Pascal, all variables have to be declared before they are used; this is different from Basic and Fortran, where the type of a variable can be determined by its name. The declaration **int i** says that **i** is an integer, precision unspecified but at least 16 bits; **long p** says that **p** is a long integer, normally 32 bits. The **printf** statement this time does more work: the first argument is a format specification (like the FORMAT statement in Fortran) that indicates how the remaining arguments are to be printed. **%d** signals an integer, **%ld** is a long integer, and \ **n** is a **newline**.

The + + operator increments by 1, so + +**i** is the same as **i**=**i** + **1**. Thus, the **for** loop shown here runs **i** from **1** to **n**, and is equivalent to Basic's

```
FOR I = 1 TO N
```

or Fortran's

```
DO I = 1, N
```

C, which began its life on a small machine, was designed for simplicity and elegance rather than features.

The function **power** is defined to have two arguments (both **int**'s), and to return a **long**; this is analogous to function declarations in Fortran or Pascal, though the syntax is different. Notice that the variable **i** in **main** is different from the **i** in **power**: Variables are local to the function in which they are declared.

Linguistic elements

Now that we have seen some C code, we can take a more organized look at the linguistic components of the language:

Control flow. Control is quite conventional, although richer than in Fortran or Basic. There are two decision-making statements, **if-else** and **switch.**

if (*expr*) *stat1* else *stat2*

expr is evaluated; if it is true (nonzero), *stat1* is executed; otherwise *stat2* is executed. The entire **else** part is optional.

```
switch (expr) {
        case const1: stat1
        case const2: stat2
        . . .
        default: stat
  }
```

expr is evaluted and its value compared against the various *const*'s. If one matches, the corresponding *stat* is executed. If there is no match, the *stat* for the **default** part is executed. **default** is optional. **switch** is like the "case" statement in Pascal, except that Pascal has no "default."

There are three loops: **while, for,** and **do.**

while (*expr*) *stat*

expr is evaluated; if it is true, *stat* is executed, and *expr* is evaluated again. When *expr* becomes false, the loop terminates.

for (*stat1*; *expr*; *stat3*) *stat2*

This is equivalent to the following **while** loop:

```
stat1
while (expr) {
        stat2
        stat3
  }
```

The **do** statement is like Pascal's "repeat-until," except for the sense of the termination test.

do *stat* while (*expr*)

stat is executed; *expr* is tested. If it is true, the loop repeats.

The statement **break** causes an immediate exit from an enclosing loop or **switch;** the statement **continue** causes the next iteration of a loop to begin. C also provides a **goto** statement, but it is infrequently used.

In all of the examples above, a *stat* can be a single statement like *x* = *3* or a group of statements enclosed in braces, which are like Pascal's "begin-end." Statements are terminated by semicolons.

Data types. Basic types in C are **char** (a single byte); **int, short,** and **long,** which are integers of different lengths; and **float** and **double,** which are floating point numbers of two different lengths. **char**'s and the various integers may be signed or unsigned.

These objects may be combined into an infinite set of "derived" data types using arrays, structures, unions, and pointers. Arrays are familiar:

```
char mesg[100];
```

defines an array **mesg** of 100 bytes, accessed as **mesg[0]** through **mesg[99].** C does not provide a string data type; arrays of **char** are used instead, with the end of the data conventionally marked by a zero byte. The compiler generates this for a string constant like **"hello world\n".** Within a string, certain "escape sequences" like **n** are used to represent special characters like **newline.** This string contains 12 characters and a terminating zero byte.

A structure is a collection of related variables that need not have the same type; it is the same idea as a record in Pascal:

```
struct object {
  int   x,y;     /* position */
  float v;       /* velocity */
  char  id[10]; /* identification */
};
struct object obj;
```

defines a structure called **object,** and declares a variable **obj** of type **object.** Individual members of the structure are referred to as **obj.v,** etc. This is essentially identical to Pascal records. Notice that the **object** structure includes an array **id,** whose components are **obj.id[0]** through **obj.id[9].** Of course, it is possible to have arrays of structures as well.

C provides pointers, or machine addresses, as an integral part of the language, in a much less restricted form than pointed in Pascal and Ada. The declarations

```
char *pc;
struct object *pobj;
```

declare **pc** a pointer to **char** and **pobj** a pointer to an object structure. The value pointed to by a structure is accessed by ***pc** or ***pobj,** as suggested by the form of the declaration. Individual structure members are accessed by, for example, **pobj**−>**v.** The "dereferencing" operation * is equivalent to the "up-arrow" or "caret" in Pascal.

If **p** is a pointer to an object of type T, and currently points to an element of an array of Ts, then **p**+**1** is a pointer to the next element of the array. That is, arithmetic operations on pointers are scaled by the size of the object to which they point. The actual size is usually irrelevant to the programmer. (When relevant, there is a **sizeof** operator that computes it, so the program doesn't contain the ex-

plicit size for any one machine.)

Operators and expressions. C has a rich set of operators compared to most conventional languages. Besides the usual arithmetic operators +, −, *, /, and % (remainder), several other groups are worth special mention. First, C provides operators for manipulating bits within a word; these are necessary for many system programming applications. These operators include

&	bitwise and
|	bitwise or
^	bitwise exclusive or
~	one's complement
<<	left shift
>>	right shift

For example, the following function counts the 1-bits in its argument by repeatedly testing the rightmost bit, then shifting the argument one position to the right until it becomes zero.

```
bitcount(n)        /*count 1 bits
                     in n */
  unsigned int n;
{
  int b;
  for (b =0; n ! = 0; n >>= 1)
    if (n & 1)
      + +b;
  return b;
}
```

The declaration **unsigned** causes **n** to be treated as a logical quantity, not an arithmetic one.

The function **bitcount** illustrates a second group of operators. Any operator such as >> that takes two operands has a corresponding "assignment operator" such as >>= so that the statement

v = v >> *expr*

can be written more concisely as

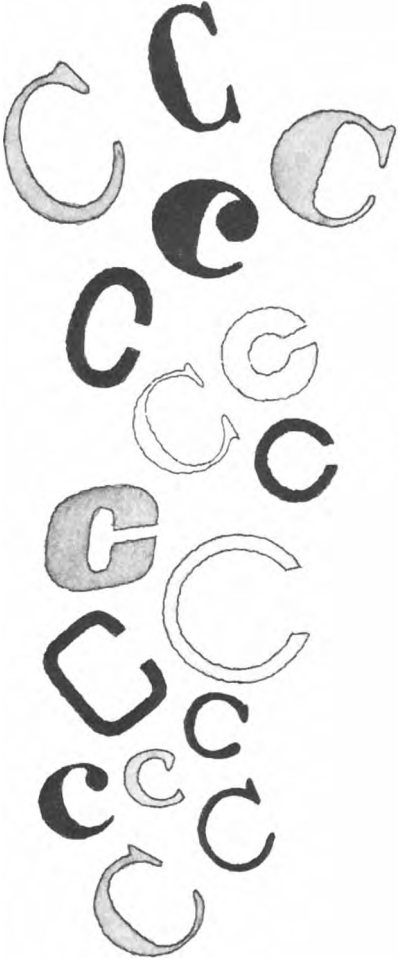
v >>= *expr*

This notation is easier to read, particularly when **v** is a complicated expression instead of a single-letter variable.

A third group of operators deals with logical conditions. The operators **&&** and | | are like .AND. and .OR. in Fortran, except that they are evaluated left to right and evaluation stops as soon as the value of the expression is known. In a construction like

```
if (i < = N && x[i] > 0) . . .
```

if **i** is greater than **N**, which is presumably the size of the array **x**, then the test involving **x[i]** will not be made. This behavior of logical operators, called "short circuit evaluation" in Ada, is highly desirable. Among the languages we have mentioned, only C and Ada provide it.



Functions. The overall structure of a C program is a set of declarations of variables and functions. These definitions are often kept in separate files if the program is large. They may be compiled separately and linked together by a linking loader. Function definitions can't be nested; this is the same as in Fortran, but a major difference from Pascal.

Within a function, variables are normally "automatic," as in Pascal: They appear when the function is entered and disappear when it is left. This was the case in the function **power** above. If a variable is declared **static,** however, it retains its value from one call to the next, as is usually the case in Fortran. Variables declared outside of any function are global—they can be referred to anywhere in the entire program.

Functions are recursive. The standard example is the factorial function:

```
fact(n)  /* returns n! (n >= 0) */
  int n;
{
  if (n <= 0)
    return 1;
  else
    return n * fact(n – 1);
}
```

The arguments to a function are passed by value, which means that the function receives a copy of the argument, not the original object. (Notice that the function **bitcount** modified its argument; this is safe because it's actually a copy.) This behavior is like the default in Pascal, but different from Fortran, where call by reference makes the original object available within the function. Since C provides pointers, it is possible to obtain the effect of call by reference when necessary by passing a pointer to the object. Function arguments and return values can be any of the basic types, pointers, structures, or unions. Arrays are passed by passing a pointer to the first element.

An assessment

Some of the material above is in part an assessment of C, particularly the comparisons with other languages. In this section, we will elaborate further.

On the positive side, it is clear that C is a compact, efficient, and expressive language. Indeed, it is good enough to have almost completely supplanted the use of assembly language programming on Unix systems. (I personally have not written an assembly program in a decade and have felt nothing but happiness at that.) The use of a clean, readable high-level language has overwhelming advantages. One is merely that it becomes possible to *read* programs, which is impossible in assembler or Basic, and excruciating in Fortran.

The path to truly efficient programs is rarely through assembly language or clever coding tricks. Efficiency comes from the selection of appropriate algorithms and data structures. For example, the bubble sort taught in introductory programming courses is astronomically slower than the Quicksort algorithm when many items are to be sorted, and no amount of low-level fiddling can close the gap.

The advantage of an expressive and readable language, such as C, is that the programmer can implement a good algorithm, and choose a suitable data structure, without being constrained by irrelevant restrictions imposed by the programming language. For instance, Quicksort is most naturally written as a recursive subroutine that passes pointers to the data being sorted. Basic does not even have subroutine arguments; recursion and pointers are foreign to both Basic and Fortran.

At the same time, although it is a high-level language, C is well endowed with facilities that permit programmers to get quite close to the machine for those situations where it is really necessary. For example, the bit-manipulation operators (shift, mask, and so on) permit logical operations without the overhead of a function call whose sole purpose is to access a particular machine instruction. Similarly, the use of pointers permits access to array elements with little of the overhead implied by conventional subscripting.

A second advantage is portability. Although it has its roots in one machine, C is free of dependencies on that machine, and so can be made portable to other machines without too much effort. As a matter of interest, a full C compiler is about 8000 lines of C. Of these, perhaps 25 percent depend on the particular machine for which code is being generated. Many of these are simply tables of code sequences. The portability of C and Unix is of great value to their users, for it frees them from the details of particular machines.

Portability refers not just to the compiler, of course, but to programs written in the language, of which the Unix system is a particularly big and important example. It is vital to portability that C is strong enough that one does not have to extend it to make it useful. By contrast, both Basic and Pascal are regularly augmented by various groups to make them suitable for tasks outside their original domains. The problem with such extensions is that programs that use them cannot be moved to other environments without considerable effort.

C has proved to be a good language for other languages to compile into. One of the best examples of that is a compiler-compiler called **yacc**, which converts the specification of a grammar for a language into a C program that is used to parse statements in that language. Naturally, one language specified this way is C itself.

Another strength of C is its absence of restrictions. A popular trend in programming languages is "strong typing," which, roughly speaking, implies that the language undertakes to check carefully that the program contains only valid combinations of data types. Pascal and Ada both provide this sort of checking, while C provides quite a bit less. Strong typing sometimes catches bugs early, but it also means that there are some programs that can't be written because they inherently require violations of the type-combination rules. A storage allocator is a good example: the Pascal "new" function, which returns a pointer to a block of storage, can't be written in Pascal because there is no way to define a function that can return an arbitrary type. However, it is easy and safe to write it in C. One measure of C's flexibility is the fact that compilers for Basic, Fortran 77, Pascal, and Ada have all been written in C. As far as I'm aware, the converse is not true.

What's wrong with C? At the lowest level, there are some poor choices of operator precedences. Some users feel that the **switch** statement should be changed so that control does not flow through from one case to the next as it does now. It is also evident that the decision to perform all floating point arithmetic in double precision is not a good idea, particularly on small machines.

There are some places where C does not provide the error checking that it could for very little effort. For example, there is no check that the arguments provided with a function call actually match what the function expects. In fact, there is no way to declare at the point of call what the arguments should be. This is likely to change, but for now it can lead to errors.

Because of the importance of maintaining compatibility with the huge body of C code in use, changes cannot be made gratuitously.

Portability problems sometimes arise because the language does not define something. For example, the order in which function arguments are evaluated is not specified, so it is possible to write code that depends on that order, and which will thus execute differently on different machines. This is not serious, since it is easy to detect the dependency, but people still overlook it from time to time, with unfortunate effects.

The future of C

C has evolved slowly through the past decade. For example, the ability to pass structures to and from functions was added about five years ago, along with an enumeration data type analogous to Pascal's enumerations. There has been a steady increase in the amount of error checking by the compiler: Although there are still few restrictions on what can be said, users are now required to say so more explicitly when they are doing something strange.

Where is C likely to go in the next few years? The most likely evolution is the continuation of this slow but steady improvement, where new features are added cautiously. Caution is necessary simply because of the importance of maintaining compatibility with the huge body of C code already in use—changes cannot be made gratuitously. One fear raised by the increasing use of C on small machines, and the number of different suppliers of compilers, is that the language definition will become blurred and different dialects will spring up. This would be a major blow, for the portability of C is one of its main strengths.

Realistically, C as it stands is not likely to change to a major degree; rather, a new language will come from it. As yet, it has not been decided whether that language should be called D or P.

For further reading

The standard reference on C is *The C Programming Language*, by B.W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). It provides a tutorial introduction, a discussion of each major language feature, and the official reference manual.

A more technical study of C as of five years ago appears in an article called "The C Programming Language," by Ritchie, Johnson, Lesk, and Kernighan, in the *Bell System Technical Journal*, July 1978. This issue is devoted to the Unix system. It contains several other papers of interest, including a discussion of the portability of C and Unix.

BCPL is still in use. It is described in a book called *BCPL: The Language and Its Compiler,"* by M. Richards and C. Whitby-Strevens (Cambridge University Press, 1979).

About the author

Brian Kernighan is head of computing structures research at Bell Laboratories and is interested in document preparation, programming languages, and programming methodology. He is the coauthor of *The C Programming Language* and *Software Tools*. □