THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The C Programming Language

By D. M. RITCHIE, S. C. JOHNSON, M. E. LESK,
and B. W. KERNIGHAN
(Manuscript received December 5, 1977)

C is a general-purpose programming language that has proven useful for a wide variety of applications. It is the primary language of the UNIX system, and is also available in several other environments. This paper provides an overview of the syntax and semantics of C and a discussion of its strengths and weaknesses.*

C is a general-purpose programming language featuring economy of expression, modern control flow and data structure capabilities, and a rich set of operators and data types.

C is not a "very high-level" language nor a big one and is not specialized to any particular area of application. Its generality and an absence of restrictions make it more convenient and effective for many tasks than supposedly more powerful languages. C has been used for a wide variety of programs, including the UNIX operating system, the C compiler itself, and essentially all UNIX applications software. The language is sufficiently expressive and efficient to have completely displaced assembly language programming on UNIX.

C was originally written for the PDP-11 under UNIX, but the language is not tied to any particular hardware or operating system. C compilers run on a wide variety of machines, including the Honeywell 6000, the IBM System/370, and the Interdata 8/32.

* UNIX is a trademark of Bell Laboratories.

I. THE LINGUISTIC HISTORY OF C

The C language in use today[1] is the product of several years of evolution. Many of its most important ideas stem from the considerably older, but still quite vital, language BCPL[2] developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B,[3] which was written by Ken Thompson in 1970 for the first UNIX system on the PDP-11.

Although neither B nor C could really be considered dialects of BCPL, both share several characteristic features with it:

(*i*) All are able to express the fundamental flow-control constructions required for well-structured programs: statement grouping, decision-making (**if**), looping (**while**) with the termination test either at the top or the bottom of the loop, and branching out to a sequence of possible cases (**switch**). It is interesting that BCPL provided these constructions in 1967, well before the current vogue for "structured programming."

(*ii*) All three languages include the concept of "pointer" and provide the ability to do address arithmetic.

(*iii*) In all three languages, the arguments to functions are passed by copying the value of the argument, and it is impossible for the function to change the actual argument. When it is desired to achieve "call by reference," a pointer may be passed explicitly, and the function may change the object to which the pointer points. Any function is allowed to be recursive, and its local variables are typically "automatic" or specific to each invocation.

(*iv*) All three languages are rather low-level, in that they deal with the same sorts of objects that most computers do. BCPL and B restrict their attention almost completely to machine words, while C widens its horizons somewhat to characters and (possibly multi-word) integers and floating-point numbers. None deals directly with composite objects such as character strings, sets, lists, or arrays considered as a whole. The languages themselves do not define any storage allocation facility beside static definition and the stack discipline provided by the local variables of functions; likewise, I/O is not part of any of these languages. All these higher mechanisms must be provided by explicitly called routines from libraries.

B and BCPL differ mainly in their syntax, and many differences stemmed from the very small size of the first B compiler (fewer

than 4K 18-bit words on the PDP-7). Several constructions in BCPL encourage a compiler to maintain a representation of the entire program in memory. In BCPL, for example,

```
valof $(
        . . .
        resultis expression
        . . .
$)
```

is syntactically an expression. It provides a way of packaging a block of many statements into a sort of unnamed internal procedure yielding a single result (delivered by the **resultis** statement). The **valof** construction can occur in the middle of any expression, and can be arbitrarily large. The B language avoided the difficulties caused by this and some other constructions by rigorously simplifying (and in some cases adjusting to personal taste) the syntax of BCPL.

In spite of many syntactic changes, B remained very close to BCPL semantically. The most characteristic feature of both languages is their nearly identical treatment of addresses (pointers). They support a model of the storage of the machine consisting of a sequence of equal-sized cells, into which values can be placed; in typical implementations, these cells will be machine words. Each identifier in a program corresponds to a cell, and a cell may contain a variety of values. Most often the value is an integer, or perhaps a representation of a character. All the cells, however, are numbered; the address of a cell is just the integer giving its ordinal position. BCPL has a unary operator **lv** (in some versions, and also in B and C, shortened to **&**) that, when applied to a name, yields the address of the cell corresponding to the name. The inverse operator **rv** (later **•**) yields the value in the cell pointed to its argument. Thus the statement

```
px = &x;
```

of B assigns to **px** the number that can be interpreted as the address of **x**; the statements

```
y = •px + 2;
•px = 5;
```

first use the value in the cell pointed to by **px** (which is the same cell as **x**) and then assign 5 to this cell.

Arrays in BCPL and B are intimately tied up with pointers. An array declaration, which might in BCPL be written

```
let Array = vec 10
```

and in B

```
auto Array[10];
```

creates a single cell named Array and initializes it with the address of the first of a sequence of 10 unnamed cells containing the array itself. Since the quantity stored in Array is just the address of the cell of the first element of the array, the expression

```
Array + i
```

is the address of the ith element, counting from zero. Likewise, applying the indirection operator,

```
* (Array + i)
```

refers to the value of the ith member of the array. This operation is so frequent that special syntax was invented to express it:

```
Array[i]
```

Thus, despite its asymmetric appearance, subscripting is a commutative operation; the above example could equally well be written

```
i[Array]
```

In BCPL and B there is only one type of object, the machine word, so when the same language operator is applied to two operands, the calculation actually carried out must always be the same. Thus, for example, if one wishes to provide the ability to do floating-point arithmetic, the "+" operator notation cannot be used, since it implies an integer addition. Instead (in a version of BCPL for the GE 635), a "." was placed in front of each operator that had floating-point operands. As may be appreciated, this was a frequent source of errors.

The machine model implied by the definitions of BCPL and B is simple and self-consistent. It is, however, inadequate for many purposes, and on many machines it causes inefficiencies when implemented. The problems became evident to us after B began to be used heavily on the first PDP-11 version of UNIX. The first followed from the fact that the PDP-11, like a number of machines (including, for example, the IBM System/370), is byte addressed; a machine address refers to any of several bytes (characters) in a word, not the word alone. Most obviously, the word orientation of B cut us off from any convenient ability to access individual bytes. Equally

important was the fact that before any address could be used, it had to be shifted left by one place. The reason for this is simple: there are two bytes per PDP-11 word. On the one hand, the language guaranteed that if 1 was added to an address quantity, it would point to the next word; on the other, the machine architecture required that word addresses be even and equal to the byte number of the first byte in the word. Since, finally, there was no way to distinguish cells containing ordinary integers from those containing pointers, the only solution visible was to represent pointers as word numbers and then, at the point of use, convert to the byte representation by multiplication by 2.

Yet another problem was introduced by the desire to provide for floating-point arithmetic. The PDP-11 supports two floating-point formats, one of which requires two words, the other four. In neither case was it satisfactory to use the trick used on the GE 635 (operators like ".+") because there was no way to represent the requirement for a single data item occupying four or eight bytes. This problem did not arise on the 635 because integers and single-precision floating-point both require only one word.

Thus the problems evidenced by B led us to design a new language that (after a brief period under the name NB) was dubbed C. The major advance provided by C is its typing structure, which completely solved the difficulties mentioned above. Each declaration in a C program specifies (sometimes implicitly) a *type*, which determines how much storage the object requires and how it is to be interpreted. The original fundamental types provided were single character (byte), integer, single-precision floating-point, and double-precision floating-point. (Others discussed below were added later.) Thus in the program

```
double a, b;
   . . .
a = b + 3;
```

the compiler is able to determine from the declarations of a and b the fact that they require four words of storage each, that the "+" means a double-precision floating add, and that "3" must be converted to floating.

Of course, the idea of typing variables is in no way original with C; in fact, it was the general rule among the most widely used and influential languages, including Algol, Fortran, and PL/I. Nevertheless, the introduction of types marked an important change in our own thinking. The typeless nature of BCPL and B had seemed to

promise a great simplification in the implementation, understanding, and use of these languages. By the time that C was created (circa 1972), advocates of languages like Algol 68 and Pascal recommended a strongly enforced type structure on psychological grounds; but even disregarding their arguments, the typeless nature of BCPL and B seemed inappropriate, for purely technological reasons, to the available hardware.

II. THE TYPE STRUCTURE OF C

The introduction of types in C, although a major departure from the tradition of BCPL and B, was done in such a way that many of the characteristic usages of the earlier languages survived. To some extent, this continuity was an attempt to preserve as much as possible of the considerable corpus of existing software written in B, but even more important, especially in retrospect, was the desire to minimize the intellectual distance between the past and the future ways of expression.

2.1 Pointers, arrays and address arithmetic

One clear example of the similarity of C to the earlier languages is its treatment of pointers and arrays. In C an array of 10 integers might be declared

```
int Array[10];
```

which is identical to the corresponding declaration in B. (Arrays begin at zero; the elements of Array are Array[0], ..., Array[9].) As discussed above, the B implementation caused a cell named Array to be allocated and initialized with a pointer to 10 otherwise unnamed cells to hold the array. In C, the effect is a bit different; 10 integers are allocated, and the first is associated with the name Array. But C also includes a general rule that, whenever the name of an array appears in an expression, it is converted to a pointer to the first member of the array. Strictly speaking, we should say, for this example, it is converted to an *integer pointer* since all C pointers are associated with a particular type to which they point. In most usages, the actual effects of the slightly different meanings of Array are indistinguishable. Thus in the C expression

```
Array + i
```

the identifier Array is converted to a pointer to the first element of

the array; i is scaled (if required) before it is added to the pointer. For a byte-addressed machine, the scale factor is the number of bytes in an integer; for a word-addressed machine the scale factor is unity. In any event, the result is a pointer to the ith member of the array. Likewise identical in effect to the interpretation of B,

```
* (Array + i)
```

is the ith member itself, and

```
Array[i]
```

is another notation for the same thing. In all these cases, of course, should Array be an array of, or pointer to, some objects other than integers, the scale factor is adjusted appropriately. The pointer arithmetic, as written, is independent of the type of object to which the pointer points and indeed of the internal representation of the pointer.

2.2 Derived types

As mentioned above, the basic types in C were originally int, which represents an integer in the basic size provided by the machine architecture; char, which represents a single byte; float, a single-precision floating-point number; and double, double-precision floating-point. Over the years, long, short, and unsigned integers have been added. In current C implementations, long is at least 32 bits; short is usually 16 bits; and int remains the "natural" size for the machine at hand. Unsigned integers exist mainly to squeeze an extra bit out of the machine, since the sign bit need not be represented.

In addition to these basic types, C provides a conceptually infinite hierarchy of derived types, which are formed by composition of the basic types with pointers, arrays, structures, unions, and functions. Examples of pointer and array declarations have already been exhibited; another is

```
double *vecp, vector[100];
```

which declares a pointer vecp to double-precision floating numbers, and an array vector of the same kind of objects. The size of an array, when specified, must always be a constant.

A *structure* is an aggregate of one or more objects, usually of various types, which can be treated as a unit. C structures are essentially the same as records in languages like Pascal, and semantically,

though not syntactically, like PL/I and Cobol structures. Thus,

```
struct tag {
        int     i;
        float   f;
        char    c[3];
};
```

defines a template, called **tag**, for a structure containing three *members*: an integer **i**, a floating point number **f**, and a three-character array **c**. The declaration

```
struct tag x, y[10], *p;
```

declares a structure **x** of this type, an array **y** of 10 such structures, and a pointer **p** to this kind of structure. The hierarchical nature of derived types is clearly evident here: **y** is an array of structures whose members include an array of characters. References to individual members of structures use the . operator:

```
x.i
x.f
y[i].c[0]
(*p).c[1]
```

Parentheses in the last line are necessary because the . binds more tightly than *. It turns out that pointers to structures are so common that special syntax is called for to express structure access through a pointer.

```
p->c[1]
p->i
```

This soon becomes more natural than the equivalent

```
(*p).c[1]
(*p).i
```

A union is capable of holding, at different times, objects of different types, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single part of storage, without embedding machine-dependent information (like the relative sizes of **int** and **float**) in a program. For example, the union u, declared

```
union {
        int     i;
        float   f;
} u;
```

can hold either an int (written u.i) or a float (written u.f). Regardless of the machine it is compiled on, it will be large enough to hold either one of these quantities. A union is syntactically identical to a structure; it may be considered as a structure in which all the members begin at the same offset. Unions in C are more analogous to PL/I's CELL than to the unions of Algol 68 or the variant records of Pascal, because it is the responsibility of the programmer to avoid referring to a union that does not currently contain an object of the implied type.

A *function* is a subprogram that returns an object of a given type:

```
unsigned unsf();
```

declares a function that returns unsigned. The type of a function ignores the number and types of its arguments, although in general the call and the definition must agree.

2.3 Type composition

The syntax of declarations borrows from that of expressions. The key idea is that a declaration, say

```
int     ... ;
```

contains a part "..." that, if it appeared in an expression, would be of type int. The constructions seen so far, for example,

```
int     *iptr;
int     ifunc();
int     iarr[10];
```

exhibit this approach, but more complicated declarations are common. For example,

```
int     *funcptr();
int     (*ptrfunc)();
```

declare respectively a function that returns a pointer to an integer, and a pointer to a function that returns an integer. The extra parentheses in the second are needed to make the * apply directly to ptrfunc, since the implicit function-call operator () binds more

tightly than *. Functions are not variables, so arrays or structures of functions are not permitted. However, a pointer to a function, like ptrfunc, may be stored, copied, passed as an argument, returned by a function, and so on, just as any other pointer.

Arrays of pointers are frequently used instead of multi-dimensional arrays. The usage of a and b when declared

```
int a[10][10];
int *b[10];
```

may be similar, in that a[5][5] and b[5][5] are both legal references to a single int, but a is a true array: all 100 storage cells have been allocated, and the conventional rectangular subscript calculation is done. For b, however, the declaration has only allocated 10 pointers; each must be set to point to an array of integers. Assuming each does point to a 10-element array, then there will be 100 storage cells set aside, plus the 10 cells for the pointers. Thus the array of pointers uses slightly more space and may require an extra initialization step, but has two advantages: it trades an indirection for a subscript multiplication, and it permits the rows of the array to be of different lengths. (That is, each element of b need not point to a 10-element vector; some may point to 2 elements, some to 20). Particularly with strings whose length is not known in advance, an array of pointers is often used instead of a multidimensional array. Every C main program gets access to its invoking command line in this form, for example.

The idea of specifying types by appropriating some of the syntax of expressions seems to be original with C, and for the simpler cases, it works well. Occasionally some rather ornate types are needed, and the declaration may be a bit hard to interpret. For example, a pointer to an array of pointers to functions, each returning an int, would be written

```
int (*(*funnyarray)[])();
```

which is certainly opaque, although understandable enough if read from the inside out. In an expression, funnyarray might appear as

```
i = (*(*funnyarray)[j])(k);
```

The corresponding Algol 68 declaration is

```
ref [] ref proc int funnyarray
```

which reads from left to right in correspondence with the informal

description of the type if ref is taken to be the equivalent of C's "pointer to." The Algol may be clearer, but both are hard to grasp.

III. STATEMENTS AND CONTROL FLOW

Control flow in C differs from other languages primarily in details of syntax. As in PL/I, semicolons are used to terminate statements, not to separate them. Most statements are just expressions followed by a semicolon; since assignments are expressions, there is no need for a special assignment statement.

Statements are grouped with braces { and }, rather than with words like begin-end or do-od, because the more concise form seems much easier to read and is certainly easier to type. A sequence of statements enclosed in { } is syntactically a single statement.

The if-else statement has the form

```
if (expression)
        statement
else
        statement
```

The *expression* is evaluated; if it is "true" (that is, if *expression* has a non-zero value), the first statement is done. If it is "false" (*expression* is zero) and if there is an else part, the second *statement* is executed instead. The else part is optional; if it is omitted in a sequence of nested if's, the resulting ambiguity is resolved in the usual way by associating the else with the nearest previous else-less if.

The switch statement provides a multi-way branch depending on the value of an integer expression:

```
switch (expression) {
        case const:
                code
        case const:
                code
        ...
        default:
                code
}
```

The *expression* is evaluated and compared against the various cases, which are labeled with distinct integer constant values. If any case

matches, execution begins at that point. If no case matches but there is a **default** statement, execution begins there; otherwise, no part of the **switch** is executed.

The **case**s are just labels, and so control may flow through one case to the next. Although this permits multiple labels on cases, it also means that in general most cases must he terminated with an explicit exit from the **switch** (the **break** statement helow).

The **switch** construction is part of C's legacy from BCPL; it is so useful and so easy to provide that the lack of a corresponding facility of acceptable generality in languages ranging from Fortran through Algol 68, and even to Pascal (which does not provide for a **default**), must he considered a real failure of imagination in language designers.

C provides three kinds of loops. The **while** is simply

```
while (expression)
        statement
```

The *expression* is evaluated; if it is true (non-zero), the *statement* is executed, and then the process repeats. When *expression* becomes false (zero), execution terminates.

The **do** statement is a test-at-the-hottom loop:

```
do
        statement
while (expression);
```

statement is performed once, then *expression* is evaluated. If it is true, the loop is repeated; otherwise it is terminated.

The **for** loop is reminiscent of similarly named loops in other languages, hut rather more general. The **for** statement

```
for (expr1; expr2; expr3)
        statement
```

is equivalent to

```
expr1;
while (expr2) {
        statement
        expr3;
}
```

Grammatically, the three components of a **for** loop are expressions. Any of the three parts can he omitted, although the semicolons must remain. If *expr1* or *expr3* is left out, it is simply dropped from

the expansion. If the test, *expr2*, is not present, it is taken as permanently true, so

```
for (;;) {
    ...
}
```

is an "infinite" loop, to be broken by other means, for example by **break**, below.

The **for** statement keeps the loop control components together and visible at the top of the loop, as in the idiomatic

```
for (i = 0; i < N; i = i+1)
```

which processes the first **N** elements of an array, the analogue of the Fortran or PL/I DO loop. The **for** is more general, however. The test is re-evaluated on each pass through the loop, and there is no restriction on changing the variables involved in any of the expressions in the **for** statement. The controlling variable **i** retains its value regardless of how the loop terminates. And since the components of a **for** are arbitrary expressions, **for** loops are not restricted to arithmetic progressions. For example, the classic walk along a linked list is

```
for (p = top; p != NULL; p = p->next)
    ...
```

There are two statements for controlling loops. The **break** statement, as mentioned, causes an immediate exit from the immediately enclosing **while**, **for**, **do** or **switch**. The **continue** statement causes the next iteration of the immediately enclosing loop to begin. **break** and **continue** are asymmetric, since **continue** does not apply to **switch**.

Finally, C provides the oft-maligned **goto** statement. Empirically, **goto**'s are not much used, at least on our system. The operating system itself, for example, contains 98 in some 8300 lines. The PDP-11 C compiler, in 9660 lines, has 147. Essentially all of these implement some form of branch to the top or bottom of a loop, or to error recovery code.

IV. OPERATORS AND EXPRESSIONS

C has been characterized as having a relatively rich set of operators. Some of these are quite conventional. For example, the basic

binary arithmetic operators are +, −, * and /. To these, C adds the modulus operator %; m%n is the remainder when m is divided by n.

Besides the basic logical or bitwise operators & (bitwise AND), and | (bitwise OR), there are also the binary operators ^ (bitwise exclusive OR), >> (right shift), and << (left shift), and the unary operator ~ (ones complement). These operators apply to all integers; C provides no special bit-string type.

The relational operators are the usual >, >=, <, <=, == (equality test), and != (inequality test). They have the value 1 if the stated relation is true, 0 if not.

The unary pointer operators * (for indirection) and & (for taking the address) were described in Section I. When y is such as to make the expressions &*y or &*y legal, either is just equal to y. Note that & and * are used as both binary and unary operators (with different meanings).

The simplest assignment is written =, and is used conventionally: the value of the expression on the right is stored in the object whose address is on the left. In addition, most binary operators can be combined with assignment by writing

$$a\ op{=}\ b$$

which has the effect of

$$a = a\ op\ b$$

except that a is only evaluated once. For example,

$$x\ +{=}\ 3$$

is the same as

$$x = x + 3$$

if x is just a variable, but

$$p[i+j+1]\ +{=}\ 3$$

adds 3 to the element selected from the array p, calculating the subscript only once, and, more importantly, requiring it to be written out only once. Compound assignment operators also seem to correspond well to the way we think; "add 3 to x" is said, if not written, much more commonly than "assign x+3 to x."

Assignment expressions have a value, just like other expressions, and may be used in larger expressions. For example, the multiple assignment

```
i = j = k = 0;
```

is a byproduct of this fact, not a special case. Another very common instance is the nesting of an assignment in the condition part of an if or a loop, as in

```
while ((c = getchar()) != EOF) ...
```

which fetches a character with the function getchar, assigns it to c, then tests whether the result is an end of file marker. (Parentheses are needed because the precedence of the assignment = is lower than that of the relational !=.)

C provides two novel operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand; the decrement operator -- subtracts 1. Thus the statement

```
++i;
```

increments i. The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++i), or postfix (after the variable: i++). In both cases, the effect is to increment i. But the expression ++i increments i *before* using its value, while i++ increments i *after* its value has been used. If i is 5, then

```
x = i++;
```

sets x to 5, but

```
x = ++i;
```

sets x to 6. In both cases, i becomes 6.

For example,

```
stack[i++] = ... ;
```

pushes a value on a stack stored in an array stack indexed by i, while

```
... = stack[--i];
```

retrieves the value and pops the stack. Of course, when the quantity incremented or decremented is a pointer, appropriate scaling is done, just as if the "1" were added explicitly:

```
*stackp++ = ... ;
... = *--stackp;
```

are analogous to the previous example, this time using a stack pointer instead of an index.

Tests may be combined with the logical connectives && (AND), || (OR), and ! (truth value negation). The && and || operators guarantee left-to-right evaluation, with termination as soon as the truth value is known. For example, in the test

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

if i is greater than N, then array[i] (presumably at that point an out-of-bounds reference) will not be accessed. This predictable behavior is especially convenient, and much preferable to the explicitly random order of evaluation promised by most other languages. Most C programs rely heavily on the properties of && and ||.

Finally, the *conditional expression,* written with the ternary operator ? :, provides an analogue of if-else in expressions. In the expression

e1 ? *e2* : *e3*

the expression *e1* is evaluated first. If it is non-zero (true), then the expression *e2* is evaluated, and that is the value of the conditional expression. Otherwise, *e3* is evaluated, and that is the value. Only one of *e2* and *e3* is evaluated. Thus to set z to the maximum of a and b,

```
z = (a > b) ? a : b;    /* z = max(a, b) */
```

We have already discussed how integers are scaled appropriately in pointer arithmetic. C does a number of other automatic conversions between data types, more freely than Pascal, for example, but without the wild abandon of PL/I. In all contexts, char variables and constants are promoted to int. This is particularly handy in code like

```
n = c - '0';
```

which assigns to n the integer value of the character stored in c, by subtracting the value of the character '0'. Generally, in fact, the basic types fall into only two classes, integral and floating-point; char variables, and the various lengths of int's, are taken to be representations of the same kind of thing. They occupy different amounts of storage but are essentially compatible. Boolean values as such do not exist; relational or truth-value expressions have value 1 if true, and 0 if false.

Variables of type int are converted to floating-point when

combined with **floats** or **doubles** and in fact all floating arithmetic is carried out in double precision, so **floats** are widened to **double** in expressions.

Conversions that involve "narrowing" an expression (for example, when a longer value is assigned to a shorter) are also well behaved. Floating point values are converted to integer by truncation; integers convert to shorter integers or characters by dropping high-order bits.

When a conversion is desired, but is not implicit in the context, it is possible to force a conversion by an explicit operator called a *cast*. The expression

(type) expression

is a new expression whose type is that specified in *type*. For example, the **sin** routine expects an argument of type **double**; in the statement

```
x = sin((double) n);
```

the value of **n** is converted to **double** before being passed to **sin**.

V. THE STRUCTURE OF C PROGRAMS

Complete programs consist of one or more files containing function and data declarations. Thus, syntactically, a program is made up of a sequence of declarations; executable code appears only inside functions. Conventionally, the run-time system arranges to call a function named **main** to start execution.

The language distinguishes the notions of *declaration* and *definition*. A declaration merely announces the properties of a variable (like its type); a definition declares a variable and also allocates storage for it or, in the case of a function, supplies the code.

5.1 Functions

The notion of *function* in C includes the subroutines and functions of Fortran and the procedures of most other languages. A function call is written

name (arglist)

where the parentheses are required even if the argument list is empty. All functions may be used recursively.

Arguments are passed by value, so the called function cannot in

any way affect the actual argument with which it was called. This permits the called program to use its formal arguments as conveniently initialized local variables. Call by value also eliminates the class of errors, familiar to Fortran programmers, in which a constant is passed to a subroutine that tries to alter the corresponding argument. An array name as an actual argument, however, is converted to a pointer to the first array element (as it always is), so the effect is as if arrays were called by reference; given the pointer, the called function can work its will on the individual elements of the array. When a function must return a value through its argument list, an explicit pointer may be passed, and the function references the ultimate target through this pointer. For example, the function swap(pa, pb) interchanges two integers pointed to by its arguments:

```
swap(px, py)        /* flip int's pointed to by px and py */
int *px, *py;
{
        int temp;

        temp = *px;
        *px = *py;
        *py = temp;
}
```

This also demonstrates the form of a function definition: the name is followed by an argument list; the arguments are declared, and the body of the function is a block, or compound statement, enclosed in braces. Declarations of local variables may follow the opening brace.

A function returns a value by

return *expression*;

The *expression* is automatically coerced to the type that the function returns. By default, functions are assumed to return int; if this is not the case, the function must be declared both in the calling routine and when it is defined. For example, a function definition is

```
double sqrt(x)      /* returns square root of x */
double x;
{
        ...
}
```

In the caller, the declaration is

```
double y, sqrt();

y = sqrt(y);
```

A function argument may be any of the basic types or a pointer, but not an array, structure, union, or function. The same is true of the value returned by a function. (The most recent versions of the language, still not standard everywhere, permit structures and unions as arguments and values of functions and allow them to be assigned.)

5.2 Data

Data declared at the top level (that is, outside the body of any function definition) are static in lifetime, and exist throughout the execution of the program. Variables declared within a function body are by default *automatic*: they come into existence when the function is entered and vanish when it is exited. Automatic variables may be declared to be **register** variables; when possible they will be placed in machine registers, which may result in smaller, faster code. The **register** declaration is only considered a hint to the compiler; no hardware register names are mentioned, and the hint may be ignored if the compiler wishes.

Static variables exist throughout the execution of a program, and retain their values across function calls. Static variables may be local to a function or (if defined at the top level) common to several functions.

External variables have the same lifetime as static, but they are also accessible to programs from other source files. That is, all references to an identically named external variable are references to the same thing.

The "storage class" of a variable can be explicitly announced in its declaration:

```
static int x;
extern double y[10];
```

More often the defaults for the context are sufficient. Inside a function, the default is **auto** (for automatic). Outside a function, at the top level, the default is **extern**. Since automatic and register variables are specific to a particular call of a particular function, they cannot be declared at the top level. Neither top-level variables nor

functions explicitly declared static are visible to functions outside the file in which they appear.

5.3 Scope

Declarations may appear either at the top level or at the head of a block (compound statement). Declarations in an inner block temporarily override those of identically named variables outside. The scope of a declaration persists until the end of its block, or until the end of the file, if it was at the top level.

Since function definitions may be given only at the top level (that is, they may not be nested), there are no internal procedures. They have been forbidden not for any philosophical reason, but only to simplify the implementation. It has turned out that the ability to make certain functions and data invisible to programs in other files (by explicitly declaring them static) is sufficient to provide one of their most important uses, namely hiding their names from other functions. (However, it is not possible for one function to access the internal variables of another, as internal procedures could do.) Similarly, the ability to conceal functions and data in one file from access by another satisfies some of the most crucial requirements of modular programming (as in languages like Alphard, CLU, and Euclid), even though it does not satisfy them all.

VI. C PREPROCESSOR

It is well recognized that "magic numbers" in a program are a sign of bad programming. Most languages, therefore, provide a way to define symbolic names for constants, so that the value of a magic number need be specified in only one place, and the rest of the code can refer to the value by some mnemonic name. In C such a mechanism is available, but it is not part of the syntax of the language; instead, symbolic naming is provided by a macro preprocessor automatically invoked as part of every C compilation. For example, given the definitions

```
#define    PI    3.14159
#define    E     2.71284
```

the preprocessor replaces all occurrences of a defined name by the corresponding defining string. (Upper-case names are normally chosen to emphasize that these are not variables.) Thus, when the programmer recognizes that he has written an incorrect value for *e*,

only the definition line has to be changed to

```
#define    E    2.71828
```

instead of each instance of the constant in the program.

Providing this service by a macro processor instead of by syntax has some significant advantages. The replacement text is not restricted to being numbers; any string of characters is permitted. Furthermore, the token being replaced need not be a variable, although it must have the form of a name. For example, one can define

```
#define    forever        for (;;)
```

and then write infinite loops as

```
forever {
    ...
}
```

The macro processor also permits macros to have arguments; this capability is heavily used by some I/O packages.

A second service of the C preprocessor is library file inclusion: a source line of the form

```
#include    "name"
```

causes the contents of the file name to be interpolated into the source at that point. (includes may be nested.) This feature is much used, especially in larger programs, for making sure that all the source files of the program are supplied with identical #defines, global data declarations, and the like.

VII. ENVIRONMENTAL CONSIDERATIONS

By intent, the C language confines itself to facilities that can be mapped relatively efficiently and directly into machine instructions. For example, writing matrix operations that look exactly like scalar operations is possible in some programming languages and occasionally misleads programmers into believing that matrix operations are as cheap as scalar operations. More important, restricting the domain of the C compiler to those areas where it knows how to do a relatively effective job provides the freedom to design subroutine libraries for the remaining tasks without constraining them to fit into some language specification. When the compiler cannot implement some facility without heavy costs in nonportability, complexity, or

efficiency, there are many benefits to leaving out such a facility: it simplifies the language and the compiler, frequently without inconveniencing the user (who often rejects a high-cost built-in operation and does it himself anyway).

At present, C is restricted to simple operations on simple data types. As a result, although the C area of operation is comparatively clean and pleasant, the user must know something about the polluting effects of the environment to get most jobs done. A program can always access the raw system calls on each system if very close interaction with the operating system is needed, but standard library routines have been implemented in each C environment that try to encourage portability while retaining speed and flexibility. The basic areas covered by the standard library at present are storage allocation, string handling, and I/O. Additional libraries and utilities are available for such areas as graphics, coroutine sequencing, execution time monitoring, and parsing.

The only automatic storage management service provided by C itself is the stack discipline for automatic variables. Two subroutines exist for more flexible storage handling. The function **calloc (n, s)** returns a pointer to a properly aligned storage block that will hold **n** items each of which is **s** bytes long. Normally **s** is obtained from the **sizeof** pseudo-function, a compile-time function that yields the size in bytes of a variable or data type. To return a block obtained from **calloc** to the free storage pool, **cfree (p)** may be called, where **p** is a value returned by a previous call to **calloc**.

Another set of routines deals with string handling. There is no "string" data type, but an array of characters, with a convention that the end of a string is indicated by a null byte, can be used for the same purpose. The most commonly used string routines perform the functions of copying one string to another, comparing two strings, and computing a string length. More sophisticated string operations can often be performed using the I/O routines, which are described next.

Most of the routines in the standard library deal with input and output. Most C programmers use stream I/O, although there is no reason why record I/O could not be used with the language. There are three default streams: the standard input, the standard output, and the error output. The most elementary routines for dealing with these streams are **getchar()** which reads a character from the standard input, and **putchar(c)**, which writes the character **c** on the standard output. In the environments in which C programs run, it is generally possibly to redirect these streams to files or other

programs; the program itself does not change and is unaware of the redirection.

The most common output function is **printf**(format, data1, data2, ...), which performs data conversion for formatted output. The string **format** is copied to the standard output, except that when a conversion specification introduced by a % character is found in **format** it is replaced by the value of the next **data** argument, converted according to the specification. For example,

```
printf("n = %d, x = %f", n, x);
```

prints n as a decimal integer and x as a floating point number, as in

```
n = 17, x = 12.34
```

A similar function **scanf** performs formatted input conversion.

All the routines mentioned have versions that operate on streams other than tbe standard input or output, and **printf** and **scanf** variants may also process a string, to allow for in-memory format conversion. Other routines in the I/O library transmit whole lines between memory and files, and check for error or end-of-file status.

Many other routines and utilities are used with C, somewbat more on UNIX than on other systems. As an example, it is possible to compile and load a C program so that when the program is run, data are collected on the number of times each function is called and how long it executes. This profile pinpoints the parts of a program that dominate the run-time.

VIII. EXPERIENCE WITH C

C compilers exist for the most widely used machines at Bell Laboratories (the IBM S/370, Honeywell 6000, PDP-11) and perbaps 10 others. Several hundred programmers within Bell Laboratories and many outside use C as their primary programming language.

8.1 Fevoreble experiences

C has completely displaced assembly language in UNIX programs. All applications code, the C compiler itself, and the operating system (except for about 1000 lines of initial bootstrap, etc.) are written in C. Although compilers or interpreters are available under UNIX for Fortran, Pascal, Algol 68, Snobol, APL, and otber

languages, most programmers make little use of them. Since C is a relatively low-level language, it is adequately efficient to prevent people from resorting to assembler, and yet sufficienctly terse and expressive that its users prefer it to PL/I or other very large languages.

A language that doesn't have everything is actually easier to program in than some that do. Tbe limitations of C often imply shorter manuals and easier training and adaptation. Language design, especially when done by a committee, often tends toward including all doubtful features, since there is no quick answer to the advocate who insists that the new feature will be useful to some and can be ignored by others. But this results in long manuals and hierarchies of "experts" who know progressively larger subsets of the language. In practice, if a feature is not used often enough to be familiar and does not complete some structure of syntax or semantics, it should probably be left out. Otherwise, tbe manual and compiler get bulky, the users get surprises, and it becomes harder and harder to maintain and use the language. It is also desirable to avoid language features that cannot be compiled efficiently; programmers like to feel that the cost of a statement is comparable to the difficulty in writing it. C has thus avoided implementing operations in the language that would have to be performed by subroutine call. As compiler technology improves, some extensions (e.g., structure assignment) are being made to C, but always with the same principles in mind.

One direction for possible expansion of the language has been explicitly avoided. Although C is much used for writing operating systems and associated software, there are no facilities for multiprogramming, parallel operations, synchronization, or process control. We believe that making these operations primitives of the language is inappropriate, mostly because language design is hard enough in itself without incorporating into it the design of operating systems. Language facilities of this sort tend to make strong assumptions about the underlying operating system that may match very poorly what it actually does.

8.2 Unfevoreble experiences

The design and implementation of C can (or could) be criticized on a number of points. Here we discuss some of the more vulnerable aspects of the language.

8.2.1 Language level

Some users complain that C is an insufficiently high-level language; for example, they want string data types and operations, or variable-size multi-dimensional arrays, or generic functions. Sometimes a suggested extension merely involves lifting some restriction. For example, allowing variable-size arrays would actually simplify the language specification, since it would only involve allowing general expressions in place of constants in certain contexts.

Many other extensions are plausible; since the low level of C was praised in the previous section as an advantage of the language, most will not be further discussed. One is worthy of mention, however. The C language provides no facility for I/O, leaving this job to library routines. The following fragment illustrates one difficulty with this approach:

```
printf("%d\n", x);
```

The problem arises because on machines on which int is not the same as long, x may not be long; if it were, the program must be written

```
printf("%D\n", x);
```

so as to tell printf the length of x. Thus, changing the type of x involves changing not only its declaration, but also other parts of the program. If I/O were built into the language, the association between the type of an expression and the format in which it is printed could be reconciled by the compiler.

8.2.2 Type safety

C has traditionally been permissive in checking whether an expression is used in a context appropriate to its type. A complete list of examples would be long, but two of the most important should illustrate sufficiently. The types of formal arguments of functions are in general not known, and in any case are not checked by the compiler against the actual arguments at each call. Thus in the statement

```
s = sin(1);
```

the fact that the sin routine takes a floating-point argument is not noticed until the erroneous result is detected by the programmer.

In the structure reference

```
p->memb
```

p is simply assumed to point to a structure of which memb is a member; p might even be an integer and not a pointer at all.

Much of the explanation, if not justification, for sucb laxity is the typeless nature of C's predecessor languages. Fortunately, a justification need no longer be attempted, since a program is now available that detects all common type mismatches. This utility, called lint because it picks bits of fluff from programs, examines a set of files and complains about a great many dubious constructions, ranging from unused or uninitialized variables through the type errors mentioned. Programs that pass unscathed through lint enjoy about as complete freedom from type errors as do Algol 68 programs, with a few exceptions: unions are not checked dynamically, and explicit escapes are available that in effect turn off checking.

Some languages, such as Pascal and Euclid, allow the writer to specify that the value of a given variable may assume only a given subrange of the integers. This facility is often connected with the usage of arrays, in that any array index must be a variable or expression whose type specifies a subset of the set given by tbe bounds of the array. This approach is not without tbeoretical difficulties, as suggested by Habermann.[4] In itself it does not solve the problems of variables assuming unexpected values or of accessing outside array bounds; such things must (in general) be detected dynamically. Still, the extra information provided by specifying the permissible range for each variable provides valuable information for the compiler and any verifier program. C has no corresponding facility.

One of the characteristic features of C is its rather complete integration of the notion of pointer and of address arithmetic. Some writers, notably Hoare,[5] have argued against the very notion of pointer. We feel, however, that the facilities offered by pointers are too valuable to give up lightly.

8.2.3 Syntax peculiarities

Some people are annoyed by the terseness of expression that is one of the characteristics of the language. We view C's short operators and general lack of noise as a benefit. For example, the use of braces { } for grouping instead of **begin** and **end** seems appropriate in view of the frequency of the operation. The use of braces even fits well into ordinary mathematical notation.

Terseness can lead to code that is hard to read, however. For example,

```
*++*argv
```

where argv has been declared char **argv (pointer into an array of character pointers) means: select the character pointer pointed at by argv (*argv), increment it by one (++*argv), then fetch the character that *that* pointer points at (*++*argv). This is concise and efficient but reminiscent of APL.

An example of a minor problem is the comment convention, which is PL/I's /* ... */. Comments do not nest, so an effort to "comment out" a section of code will fail if that section contains a comment. And a number of us can testify that it is surprisingly hard to recognize when an "end comment" delimiter has been botched, so that the comment silently continues until the next comment is reached, deleting a line or two of code. It would be more convenient if a single unique character were reserved to introduce a comment, and if comments always terminated at an end of line.

8.2.4 Semantic peculiarities

There are some occasionally surprising operator precedences. For example,

```
a >> 4 + 5
```

shifts right by 9. Perhaps worse,

```
(x & MASK) == 0
```

must be parenthesized to associate the proper way. Users learn quickly to parenthesize such doubtful cases; and when feasible lint warns of suspicious expressions (including both of these).

We have already mentioned the fact that the case actions in a switch flow through unless explicitly broken. In practice, users write so many switch statements that they become familiar with this behavior and some even prefer it.

Some problems arise from machine differences that are reflected, perhaps unnecessarily, into the semantics of C. For example, the PDP-11 does sign extension on byte fetches, so that a character (viewed arithmetically) can have a value ranging from −128 to +127, rather than 0 to +255. Although the reference manual makes it quite clear that the precise range of a char variable is machine dependent, programmers occasionally succumb to the

temptation of using the full range that their local machine can represent, forgetting that their programs may not work on another machine. The fundamental problem, of course, is that C permits small numbers, as well as genuine characters, to be stored in char variables. This might not be necessary if, for example, the notion of subranges (mentioned above) were introduced into the language.

8.2.5 Miscellaneous

C was developed and is generally used in a highly responsive interactive environment, and accordingly the compiler provides few of the services usually associated with batch compilers. For example, it prepares no listing of the source program, no cross reference table, and no indication of the nature of the generated code. Such facilities are available, but they are separate programs, not parts of the compiler. Programmers used to batch environments may find it hard to live without giant listings; we would find it hard to use them.

IX. CONCLUSIONS AND FUTURE DIRECTIONS

C has continued to develop in recent years, mostly by upwardly compatible extensions, occasionally by restrictions against manifestly nonportable or illegal programs that happened to be compiled into something useful. The most recent major changes were motivated by the extension of C to other machines, and the resulting emphasis on portability. The advent of union and of casts reflects a desire to be more precise about types when moving to other machines is in prospect. These changes have had relatively little effect on programmers who remained entirely on the UNIX system. Of more importance was a new library, which changed the use of a "portable" library from an option into an effective standard, while simultaneously increasing the efficiency of the library so that users would not object.

It is more difficult, of course, to speculate about the future. C is now encountering more and more foreign environments, and this is producing many demands for C to adapt itself to the hardware, and particularly to the operating systems, of other machines. Bit fields, for example, are a response to a request to describe externally imposed data layouts. Similarly, the procedures for external storage allocation and referencing have been made tighter to conform to requirements on other systems. Portability of the basic language

seems well handled, but interactions with operating systems grow ever more complex. These lead to requests for more sophisticated data descriptions and initializations, and even for assembler windows. Further changes of this sort are likely.

What is not likely is a fundamental change in the level of the language. Realistically, the very acceptance of C has compelled changes to be made only most cautiously and compatibly. Should the pressure for improvements become too strong for the language to accommodate, C would probably have to be left as is, and a totally new language developed. We leave it to the reader to speculate on whether it should be called D or P.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
2. M. Richards, "BCPL: A Tool for Compiler Writing and Systems Programming," Proc. AFIPS SJCC, *34* (1969), pp. 557-566.
3. S. C. Johnson and B. W. Kernighan, "The Programming Language B," Comp. Sci. Tech. Rep. No. 8, Bell Laboratories (January 1973).
4. A. N. Habermann, "Critical Comments on the Programming Language PASCAL," Acta Informatica, *3* (1973), pp. 47-58.
5. C. A. R. Hoare, "Data Reliability," ACM SIGPLAN Notices, *10* (June 1975), pp. 528-533.