AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

The Evolution of C—Past and Future

By L. ROSLER*

(Manuscript received September 12, 1983)

The C programming language was developed originally to implement *UNIX*™ operating systems and their utilities. It has become a mainstay of systems and application programming at AT&T Bell Laboratories, and is rapidly growing in commercial importance. It continues to evolve in response to the needs of new environments, spanning the range from tiny peripheral controllers to huge electronic switching systems written and maintained by hundreds of programmers. There are severe reliability and real-time constraints throughout this spectrum. This paper reports changes made so far to meet the needs of these new environments and indicates the directions of current developments.

I. INTRODUCTION

The C programming language was designed in the early 1970's by Dennis M. Ritchie as part of the development of the original *UNIX* operating system.[1] The capabilities of the language for programming portable operating systems were enhanced rapidly as the first *UNIX* system was ported to other processors.

In 1978 Kernighan and Ritchie published the definitive description and reference manual[2] for the C programming language as it existed then. They were joined by Johnson and Lesk in a descriptive article[3]

* AT&T Bell Laboratories.

in this journal that evaluated the language after five years of experience, and projected future directions for its growth. This paper reports changes made in the succeeding years and indicates the direction of current developments.

A major trend in the development of C is toward stricter type checking, along the lines of languages like Pascal.[4] However, in accordance with what has been called the "spirit" of C (meaning a model of computation that is close to that of the underlying hardware), many areas of the language specification deliberately remain permissive. This allows implementors the freedom to achieve maximum efficiency by using the instructions most appropriate for each machine. (For example, the sign of the remainder on a division involving negative integers is explicitly unspecified.)

In keeping with the original sparse design of the language, nothing has been added that can only be implemented effectively by calling a run-time function. (This does not prevent an implementor from choosing to implement an operation in the language for which the hardware support is inadequate by a call to a hidden function. For example, this may be the most appropriate way to implement floating-point arithmetic on processors that do not support floating-point operations.) For this reason, the exponentiation operation is not part of the language, but must be explicitly invoked by the programmer as a function in the library.

Many other capabilities (including input/output, storage allocation, and mathematics) are integral parts of other languages but not of C. For practical reasons of application portability, the libraries that provide these capabilities for C are also subject to standardization, so they now might reasonably be viewed as extensions of the language. In recent years, major enhancements in functionality and efficiency were made to these standard support libraries. However, this paper will focus on the language proper.

Note that the material presented here represents changes to the AT&T Bell Laboratories definition of the language, not to any implementation. No existing compiler fully implements the new definition as yet, which is itself subject to change as a result of standardization efforts.

The reader is presumed to have some familiarity with C as presented by Kernighan and Ritchie.[2] References in parentheses refer to sections in *The C Reference Manual* printed as Appendix A of that book. However, this paper can be understood without having the book at hand.

II. PORTABILITY AND STANDARDS

To maintain the stability of a mature language while allowing

controlled evolution is both a technical and an administrative challenge.

Since 1977, the Computer Technologies Area of AT&T Bell Laboratories has sponsored a committee to develop and maintain internal C standards. This committee monitors and promotes the portability and evolution of the C language proper, the support libraries without which useful work in C is impossible, and the many *UNIX* systems and other environments in which C is implemented. As a result of that effort, applications that do not rely heavily on the characteristics of the supporting hardware or operating system can be moved from one environment to another without significant reprogramming.

In recognition of the growing commercial importance of C, the American National Standards Institute (ANSI) chartered a technical committee (X3J11) to develop a standard for the language, libraries, and environment. The current schedule calls for a draft to be published for public comment early in 1985.

III. MANAGING INCOMPATIBLE CHANGES

Inevitably, some of the changes that were made alter the semantics of existing valid programs. Those who maintain the various compilers used internally try to ensure that programmers have adequate warning that such changes are to take effect, and that the introduction of a new compiler release does not force all programs to be recompiled immediately.

For example, in the earliest implementations the ambiguous expression `x =-1` was interpreted to mean "decrement x by 1". It is now interpreted to mean "assign the value -1 to x". This change took place over the course of three annual major releases. First, the compilers and the `lint` program verifier[5] were changed to generate a message warning about the presence of an "old-fashioned" assignment operator such as `=-`. Next, the parsers were changed to the new semantics, and the compilers warned about an ambiguous assignment operation. Finally, the warning messages were eliminated.

Support for the use of an "old-fashioned initialization"

```
int x 1;
```

(without an equals sign) was dropped by a similar strategy. This helps the parser produce more intelligent syntax-error diagnostics.

Predictably, some C users ignored the warnings until introduction of the incompatible compilers forced them to choose between changing their obsolete source code or assuming maintenance of their own versions of the compiler. But on the whole the strategy of phased change was successful.

IV. SIGNIFICANT CHANGES

The changes discussed in this section represent significant shifts in the orientation and capabilities of the language. Unless we explicitly state it, all the changes described are backward-compatible.

4.1 Float and double

In the arena of the original application of C (the implementation of *UNIX* systems), the efficiency of floating-point arithmetic was of little importance. Support libraries were simpler if only one type of value was handled. Furthermore, the hardware of the first production implementation favored the use of double precision over single precision.

These considerations manifested themselves as a requirement that all floating-point arithmetic be done in double precision (Ref. 2, Sect. 6.2). In addition to providing a marginally useful increase in default accuracy, this choice helped keep the code generators simple.

This requirement now seems inappropriate, in view of the following changed circumstances:

1. Because of its other desirable attributes, C is being used more frequently in areas such as scientific calculation, where computationally oriented languages such as Fortran were the traditional choices. A general-purpose language should support floating-point arithmetic as efficiently as possible.
2. In fact, most implementations perform double-precision arithmetic more slowly than single-precision, and access to the operands is more costly.
3. Many code generators for C are enhanced to share support for languages (such as Fortran) that *require* single-precision arithmetic in a single-precision context.

Therefore, C compilers may now use single-precision operations to implement floating-point arithmetic that involves single-precision operands. Interfunction linkages (arguments, formal parameters, and return values) declared to be `float` are still coerced implicitly to `double`. This resembles the widening of `char` and `short` arguments to `int`, and simplifies the maintenance of libraries and the specification of constants as arguments. The called function can declare the formal parameter as `float` if desired.

4.2 Type specifiers

4.2.1 Void

Unlike many other languages, C makes no syntactic distinction between procedures that return a value (functions) and procedures that have only side effects (subroutines). Both are called functions in C.

Because most useful functions do return values, in particular integer values in most systems programming environments, the language

permits the declaration for a function returning an integer to be omitted (Ref. 2, Sect. 13). Furthermore, even if a declaration is given, for example:

```
extern f( );
```

if no type is specified it is taken to be integer (Ref. 2, Sect. 8.2).

This convenient default leads to various incorrect descriptions regarding functions that in fact return no value. For example, how could one declare a pointer to such a function? As some type must be specified:

```
int (*fp)( ) = f;
```

the declaration is interpreted as a pointer to a function returning an integer, even though no value is in fact returned.

The new type `void` has been added to deal with this anomaly. It can be used only to declare a function that returns no value or as a cast to state explicitly that the value returned by a function is being ignored. Obviously, the nonexistent "value" of a function declared as returning `void` cannot be used in an expression or cast to any other type.

4.2.2 Enum

An enumeration data type has been added to C. It is similar in intent to the enumerated type of Pascal—to restrict the set of values that can be assigned to specific integer variables. In the following example

```
enum fruit {apple, orange, pear} lunch, dinner;
```

`lunch` and `dinner` are integer variables that have assigned to them only the values `apple`, `orange`, or `pear`. The optional tag `fruit` may be used to refer to this enumeration elsewhere.

A significant difference from Pascal is that values may be specified for any or all of the integer constants that constitute an enumeration:

```
enum permissions {read = 4, write = 2, execute = 1};
```

A value may even be duplicated:

```
enum unities {one = 1, uno = 1, eins = 1, odin = 1};
```

The name of an enumeration constant may not be reused in a different enumeration, however, even with the same value.

The successor, predecessor, and ordinal functions of Pascal are not available. Therefore, it is not possible in C to write a simple loop over the values of an enumeration variable, because they need not form a linear sequence.

Enumeration constants provide a convenient way of moving into the compiler proper a task that could be handled in the preprocessor by a list of `#define` names. This helps in symbolic debugging, as the identifiers themselves appear in the symbol table. It also eliminates the need to supply sequential values that may in themselves have no interest.

4.3 Structures and unions

4.3.1 Names of members

In the original specification (Ref. 2, Sect. 8.5), all members of structures in a single compilation had to have unique names. The only exception was that the same name could be used in two different structures if the type and offset were the same in both.

Because of the likelihood of name conflicts in large applications (where header files might include several hundred structure definitions), these rules were relaxed to allow the same name to be used in more than one structure or union, even with different types or offsets. For this to be effective, any reference to a structure or union member must be fully qualified, and the type of reference must be the same as the type of structure or union containing the member referred to.

In other words, it is no longer valid to refer to one type of structure using a pointer declared as pointing to another type of structure, or using an integer as a pointer. An explicit cast must be used. This closes a previous loophole (Ref. 2, Sect. 14.1) and is not backward-compatible. (Type equivalence is name equivalence—structures with different tags are of different types, even if their members are identical.)

This major change was introduced in phases, in the same way as the change from `=op` to `op=` described in an earlier section. Compiler warnings identified incomplete qualifications and type conflicts, but the programs could still be compiled unambiguously, as the names of members all had to be unique to begin with.

4.3.2 Assignments, parameters, and function values

As Ref. 2, Sect. 14.1 predicts, the semantics of structures and unions has been enriched. The value of a structure or union may be assigned to another one of the same type; a structure or union may be passed as an argument to a function; and a function may return a structure or union as its value. For example:

```
struct s a, b, f( );
a = b; a = f(b);
```

are valid declarations and statements.

Even though similar operations on arrays exist in other languages,

these desirable enhancements could not be retrofitted to arrays in C. The interpretation of an array name as a pointer expression is embedded too deeply in existing programs (Ref. 2, Sect. 7.1).

V. OTHER CHANGES

These changes are presented here in the order of the relevant sections in *The C Reference Manual.* They also are backward-compatible, except as described.

5.1 Lexical conventions

Form feeds and vertical tabs are added to the list of characters (Ref. 2, Sect. 2) that serve as "white space" to separate tokens and "line breaks" for compiler control lines. No semantics had previously been ascribed to these characters.

5.2 Key words

As we discussed above, two new key words, `void` and `enum`, were added to represent new types. This change affects only programs that happened to use those words as identifiers.

The `entry` key word (Ref. 2, Sect. 2.3) was never implemented and is no longer reserved.

5.3 Constants

The digits `8` and `9` are no longer accepted in octal-integer constants (Ref. 2, Sect. 2.4.1). Though not backward-compatible, this change had little impact, as few programmers used this quirk in writing octal constants.

Previously, the backslash in an undefined escape sequence in a character or string constant was explicitly ignored (Ref. 2, Sect. 2.4.3), so that `'\z'`, for example, was a strange but acceptable way of writing `'z'`. Now, the meaning of an undefined escape sequence is explicitly undefined, so `'\z'` has no meaning.

This too is an incompatible change, but is justifiable since it allows new escape sequences to be defined in the future without affecting existing valid programs. As an example, the escape sequence `\v` has been added to denote a vertical tab. A proposal has been adopted to use the escape sequence `\xddd` to describe a hexadecimal constant, analogous to the existing `\ddd` notation for an octal constant.

5.4 Initialization

Arbitrary restrictions in any area of a language are undesirable, since they add to the difficulty of learning and using it.

The restriction against initializing an automatic array or structure (Ref. 2, Sect. 8.6) was based on practical considerations of compiler

complexity, not on theoretical objections. This restriction has been removed, though no compiler yet implements this capability. The syntax is identical to that used for initializing an external or static array or structure.

The restriction against initializing a union was based on the lack of suitable unambiguous syntax. The ANSI draft standard will propose that a union be initialized according to the type of the first member in its declaration, ascribing for the first time significance to the order of declaration.

With these changes, there will no longer be any object that cannot be initialized.

5.5 Type specifiers

Every size of integer now has a corresponding `unsigned` type (Ref. 2, Sect. 8.2).

In anticipation of the extension of C to support more than two sizes of floating-point numbers (in accordance with a proposed IEEE standard[6]), the type `long float` is no longer accepted as a synonym for `double`. This change should have minimal impact on existing programs, as the synonym seems to have been used infrequently, if at all.

5.6 Defined type

Even though in a construction such as

```
typedef int KILOMETERS;
KILOMETERS distance;
```

the type of `distance` is `int` (Ref. 2, Sect. 8.8), the defined type may not be further modified by `long`, `short`, or `unsigned`. For example,

```
long KILOMETERS to_the_moon;
```

is invalid; a new type must be defined:

```
typedef long int ASTRONOMICAL;
ASTRONOMICAL to_the_moon;
```

This is a clarification, not a change.

5.7 Switch statement

The restriction that the controlling expression of a switch statement have type `int` (Ref. 2, Sect. 9.2) is being removed. Any integral type will be permitted, and the case-expressions will be coerced to that type.

5.8 External data definitions

Of all the areas of potential change, this has caused the most

controversy. The manual states (Ref. 2, Sect. 10.2) that the default storage class for an external data definition is `extern`. Thus, when several external data definitions of the form `int i` appear, the intention is to define a single variable, *i*, whether or not the `extern` key word is present.

This implies the existence of a mechanism similar to that of Common in Fortran, which associates multiple definitions of the same external identifier. Limitations in the support software in several vendor-supplied operating systems make it difficult or impossible to implement this design intent. Therefore, a distinction was introduced (Ref. 2, Sect. 11.2) in the use of the `extern` key word – its appearance indicated a declaration for the external variable in question, its absence indicated a definition. Most important of all, there has to be exactly one such definition in the set of files constituting a single program.

Thus this restriction is actually a portability constraint imposed by some environments, not a characteristic of the C language itself. The capability of many *UNIX* system implementations to allow more than one identical external data definition to appear (without the `extern` key word) is considered to be an extension to the more restrictive ANSI draft standard.

5.9 Compiler control lines

The conditional-compilation facility (Ref. 2, Sect. 12.3) has been enhanced in two ways.

To facilitate selection of one among a set of choices, any number of control lines of the form

`#elif` *constant-expression*

may now appear between a `#if` line and its closing `#endif` (or `#else` if present).

The new pseudofunction `defined` (*identifier*) may be used in the *constant-expression* part of a `#if` or `#elif` control line, with value 1 if the identifier is currently defined in the preprocessor, and 0 otherwise. Thus, `#ifdef` *identifier* is equivalent to `#if defined` (*identifier*), and `#ifndef` *identifier* is equivalent to `#if !defined` (*identifier*). The older forms will be retained for backward compatibility, as they are deeply entrenched in existing code. But, as they are superfluous, equivalents to `#ifdef` will not be provided for the new construction `#elif`.

VI. INTRACTABLE PROBLEMS

6.1 Preprocessing

One unfortunate effect of preprocessing the text before compilation is that programmers must know which functions are macroinstruc-

tions. They may not be declared; they do not obey the call-by-value semantics of C functions; and their arguments may be evaluated an unknown number of times, so side effects are unpredictable. A general trend for the future will be to rely less on the preprocessor and more on the compiler.

6.2 Integer sizes

Although the portability of C has been amply demonstrated over the past decade,[5,7] persistent problems arise where the size of a `long int` differs from that of an ordinary `int`.

For example, the difference of two pointers has been described as an ordinary `int` (Ref. 2, Sect. 7.4). But in a large-address environment where a pointer has the same size as a `long int` (Ref. 2, Sect. 14.4), an ordinary `int` may not be large enough to store the difference. This would impose an arbitrary limit on the size of an array. It is now agreed that the difference should have the same size as the pointers being subtracted.

This solves the problem only in part. Consider the common situation where the difference is used, for example, as an argument to an input/output function. Such an argument cannot be declared portably, but a suitable type definition could be provided as part of a standard header file.

VII. FUTURE DIRECTIONS

All the enhancements and changes to the language defined by Kernighan and Ritchie[2] discussed in the preceding sections exist in many widely used compilers and have been presented to the ANSI X3J11 committee for standardization. The section that follows deals with later proposals that are still being evaluated.

One major proposed enhancement, the introduction of classes (abstract data types) similar to those of Simula, is presented in a companion article.[8] Other enhancements, presented in this section, are in use internally, but have not yet been exposed to large numbers of programmers. They are reported here to indicate some of the anticipated directions of language evolution.

7.1 Argument typing

At present, most C compilers make no checks on the number and type consistency of function invocations, even within a single compilation. In *UNIX* systems, this responsibility is delegated to the `lint` program verifier, which checks, among many other things, the consistency of function interfaces over an entire program set and associated libraries.

Because of the computer resources required to do the extra parsing

involved, the cost of using `lint` in the development of very large programs may be prohibitive. User-generated `lint` libraries that declare function arguments and return values but omit function bodies relieve this cost somewhat, but must be kept in phase with the real source. It would be better to provide a way, as part of a function declaration, for the compiler itself to be informed of not only the type returned by the function (as at present), but also of the types of the function arguments.

A method has been developed to do this in a backward-compatible way.[9] In a function declaration, arguments may be declared sequentially by type, thus:[†]

```
char *fgets(char *, int, FILE *);
```

When no further information about the arguments is provided, a trailing comma is added:

```
int fscanf(FILE *, char *,);
```

When no information at all about the arguments is provided, nothing is between the parentheses, which is compatible with existing programs. The special case of declaring a function with no arguments is handled via the `void` key word:

```
int rand(void);
```

Perhaps the most important payoff of argument typing is that, if possible, an argument is coerced to the type of its corresponding formal parameter, as if by assignment. This will eliminate a major source of interface errors in large programs. Incompatibilities (such as an integer argument and a pointer formal parameter) will cause fatal compilation errors.

7.2 The "const" type specifier

A new type specifier, `const`, has been added[9] to meet a need that has long been recognized—declaring that the value with which a particular variable is initialized may not be changed during execution of the program.

In some environments, this may simply tell the compiler not to allow the variable to appear on the left-hand side of an assignment and not to allow its address to be assigned to a pointer through which it may be modified. (Such an implementation could not protect against an inadvertent modification caused by a wild pointer.) This is the most protection that can be provided if the `const` variable has `auto`

[†] The examples are functions in the Standard Library, partly described in Chapter 7 of Kernighan and Ritchie.[2]

or `register` storage class, so that it is initialized dynamically on each entry to the block in which it is defined, and the value with which it is initialized is itself variable.

If the storage class of the variable is `extern` or `static`, or if the initializer is constant, the compiler may be able to place the data in an area of memory protected by hardware against modification. This also allows space to be saved by sharing the data among several simultaneous executions of the program, just as the program text may be shared in some implementations. The data may even be placed into read-only memory if desired.

This mechanism is particularly appropriate for large arrays of permanent data, such as parse tables or constant character strings. To achieve the desired end, some programmers have resorted to editing the assembly language produced by current compilers. At the cost of reserving yet another key word (possibly used as an identifier in existing programs), this new facility legitimizes the needed capability in the language proper.

An interesting distinction can be made between pointers that themselves are constant:

`char * const` *constant-pointer*

and pointers to constant data:

`const char *` *pointer-to-constant*

The latter can be used to declare that even though an argument is a pointer the function does not change the data pointed to.

```
char *strcpy(char *, const char *);
```

declares that `strcpy` gets two arguments that are character pointers, but does not change the array pointed to by the second argument.

7.3 Assembler windows

Access to the hardware of the operating environment is often requested. Code for implementing operating systems or device drivers may need to manipulate particular registers or to execute instructions that are inaccessible from C but accessible through the assembly language of the machine.

Assembly language may also be needed for efficiency. For example, C does not support the assignment of one array or string to another, and the programmer must write a loop to do this operation one element at a time. Yet many machines have extremely efficient implementations for block moves.

The need for access to special hardware is recognized by providing standardized library functions, which may be implemented either in

C or in assembly language as appropriate to a particular environment. But, in time-critical applications, even the overhead of function linkage may be too high.

Therefore, the need has long been felt for the ability to interject instructions in assembly language directly in the midst of C code. The use of such a mechanism destroys portability, and may interfere with analysis or optimization of the function containing the alien statements.

Many existing C compilers use the key word `asm` for this purpose. A statement of the form

`asm` (*string*);

causes the specified string to be injected directly into the assembly-language output of the compiler.

This capability is still not powerful enough for many applications. No access is provided to identifiers in the C program, so the programmer may have to make assumptions about which registers should be addressed by the assembly-language statements.

An experimental implementation now being evaluated uses the key word `asm` in a different context.[10] A declaration of the form

`asm` *f* (*arg1*, *arg2*, · · ·) {· · ·}

defines a function *f* to be compiled in line (without function linkages). The programmer can specify alternate assembly-language expansions in the function prototype, depending on the storage classes of the actual parameters.

VIII. EVOLUTIONARY STUBS

By no means have all the experimental enhancements made to C been accepted as part of the official language. Many developers have tried to enrich the syntax of the language to individual tastes, but these efforts did not win wide support. This section describes one evolutionary stub of more substantial significance, which though it did not lead to changes in C did provide valuable insight into an important problem in the development of large programs by many programmers.

In a very large multifile C program, it is difficult to control the scopes of external definitions except by carefully structuring a multiplicity of header files and including them selectively in the various compilation units. One project tried a different solution to this problem: introducing new preprocessor directives to export explicitly the definitions of specific variables to other files and to import the declarations from other files. To eliminate unnecessary compilation, a

program automatically generated files describing the dependencies for use by the `make` utility,[11] or its enhancement, the `build` utility.[12]

This attempt foundered because of the need to create and maintain hidden interface files separate from the source files. This arose because of the possibility of circular dependencies between the variables in several files. The solution to this problem—explicitly separating the external interfaces from the program text and managing the dependencies using a database manager—is now part of the Ada† language and programming support environment.

The valuable idea of generating "makefiles" automatically by analyzing the inclusions of header files is being incorporated in other tools, however.

IX. SUMMARY

In its decade of existence, C grew beyond its original conception as a language for implementing operation systems into a full general-purpose language. This was accomplished by small changes, mostly backward-compatible, that have not fundamentally altered the original sparse design.

A major trend in the development of the language is toward stricter type checking, particularly in the use of pointers and in function argument type checking. On the other hand, the model of computation remains close to that of the underlying hardware.

Though mature, the C language continues to evolve in a controlled way. Internal and external standardization activities will continue to impose requirements for backward compatibility in the future.

X. ACKNOWLEDGMENTS

Dennis M. Ritchie, the author of C, continues to be closely associated with its evolution and standardization. His perceptive observations and insights over the years are greatly appreciated.

Many colleagues provided useful comments on drafts of this paper. I particularly thank Bjarne Stroustrup, whose ideas are strongly influencing the future evolution of the language. Lively discussions among the members of the X3J11 committee have helped clarify many potential misinterpretations of the language specification.

Because of their potential value, the proposals described in Sections 7.1 (argument typing) and 7.2 (`const`) have been accepted by the X3J11 committee.

† Trademark of the U.S. Department of Defense, Ada Joint Program Office.

REFERENCES

1. D. M. Ritchie and K. L. Thompson, *The UNIX Time-sharing System*, Commun. ACM, *17*, No. 7 (July 1974), pp. 365–75.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
3. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 1991–2020.
4. *IEEE Standard Pascal Computer Programming Language*, New York: The Institute of Electrical and Electronics Engineers, Inc., 1983.
5. S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the *UNIX* System," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 2021–48.
6. Draft 10.0 of IEEE Task P754, *A Proposed Standard for Binary Floating-Point Arithmetic*, December 2, 1982.
7. L. Rosler, "The Best of *UNIX* on GCOS," Honeywell Large Systems Users' Association, October 1978.
8. B. Stroustrup, "The *UNIX* System: Data Abstraction in C," AT&T Bell Lab. Tech. J., this issue.
9. B. Stroustrup, private communication.
10. R. J. Mascitti, private communication.
11. S. I. Feldman, "Make—A Program for Maintaining Computer Programs," Software Practice and Experience, *9*, No. 4 (April 1979), pp. 255–65.
12. V. B. Erickson and J. F. Pellegrin, "Build—A Software Construction Tool," AT&T Bell Lab. Tech. J., *63*, No. 6, Part 2 (July–August 1984), pp. 1049–59.

AUTHOR

Lawrence Rosler, A.B. (Physics), 1953, Cornell University; M.S., Ph.D. (Physics), Yale University, in 1954 and 1958, respectively; AT&T Bell Laboratories, 1957—. Mr. Rosler is Supervisor of the Language Systems Engineering group in the *UNIX* Languages and Programming Environment Development department. His early work involved the development of solid-state electronic devices. His more recent work includes the design of languages for interactive graphics terminals, the implementation of the C language and libraries on various systems, and the management of C language development for *UNIX* systems. Chairman, Language Subcommittee, American National Standards Institute Technical Committee X3J11 for the Programming Language C; member, ACM, APS, Sigma Xi, AAAS.