The Development of the C Language

Dennis M. Ritchie
Bell Labs/Lucent Technologies
Murray Hill, NJ 07974 USA

dmr@bell-labs.com

ABSTRACT

The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today. This paper studies its evolution.

Introduction

This paper is about the development of the C programming language, the influences on it, and the conditions under which it was created. For the sake of brevity, I omit full descriptions of C itself, its parent B [Johnson 73] and its grandparent BCPL [Richards 79], and instead concentrate on characteristic elements of each language and how they evolved.

C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972. Another spate of changes peaked between 1977 and 1979, when portability of the Unix system was being demonstrated. In the middle of this second period, the first widely available description of the language appeared: *The C Programming Language,* often called the 'white book' or 'K&R' [Kernighan 78]. Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the computer industry.

History: the setting

The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories [Ritchie 78] [Ritchie 84]. The company was pulling out of the Multics project [Organick 75], which had started as a joint venture of MIT, General Electric, and Bell Labs; by 1969, Bell Labs management, and even the researchers, came to believe that the promises of Multics could be fulfilled only too late and too expensively. Even before the GE-645 Multics machine was removed from the premises, an informal group, led primarily by Ken Thompson, had begun investigating alternatives.

Thompson wanted to create a comfortable computing environment constructed according to his own design, using whatever means were available. His plans, it is evident in retrospect,

 This article was presented at Second History of Programming Languages conference, Cambridge, Mass., April, 1993.

incorporated many of the innovative aspects of Multics, including an explicit notion of a process as a locus of control, a tree-structured file system, a command interpreter as user-level program, simple representation of text files, and generalized access to devices. They excluded others, such as unified access to memory and to files. At the start, moreover, he and the rest of us deferred another pioneering (though not original) element of Multics, namely writing almost exclusively in a higher-level language. PL/I, the implementation language of Multics, was not much to our tastes, but we were also using other languages, including BCPL, and we regretted losing the advantages of writing programs in a language above the level of assembler, such as ease of writing and clarity of understanding. At the time we did not put much weight on portability; interest in this arose later.

Thompson was faced with a hardware environment cramped and spartan even for the time: the DEC PDP-7 on which he started in 1968 was a machine with 8K 18-bit words of memory and no software useful to him. While wanting to use a higher-level language, he wrote the original Unix system in PDP-7 assembler. At the start, he did not even program on the PDP-7 itself, but instead used a set of macros for the GEMAP assembler on a GE-635 machine. A postprocessor generated a paper tape readable by the PDP-7.

These tapes were carried from the GE machine to the PDP-7 for testing until a primitive Unix kernel, an editor, an assembler, a simple shell (command interpreter), and a few utilities (like the Unix *rm, cat, cp* commands) were completed. After this point, the operating system was self-supporting: programs could be written and tested without resort to paper tape, and development continued on the PDP-7 itself.

Thompson's PDP-7 assembler outdid even DEC's in simplicity; it evaluated expressions and emitted the corresponding bits. There were no libraries, no loader or link editor: the entire source of a program was presented to the assembler, and the output file with a fixed name that emerged was directly executable. (This name, *a.out*, explains a bit of Unix etymology; it is the output of the assembler. Even after the system gained a linker and a means of specifying another name explicitly, it was retained as the default executable result of a compilation.)

Not long after Unix first ran on the PDP-7, in 1969, Doug McIlroy created the new system's first higher-level language: an implementation of McClure's TMG [McClure 65]. TMG is a language for writing compilers (more generally, TransMoGrifiers) in a top-down, recursive-descent style that combines context-free syntax notation with procedural elements. McIlroy and Bob Morris had used TMG to write the early PL/I compiler for Multics.

Challenged by McIlroy's feat in reproducing TMG, Thompson decided that Unix possibly it had not even been named yet needed a system programming language. After a rapidly scuttled attempt at Fortran, he created instead a language of his own, which he called B. B can be thought of as C without types; more accurately, it is BCPL squeezed into 8K bytes of memory and filtered through Thompson's brain. Its name most probably represents a contraction of BCPL, though an alternate theory holds that it derives from Bon [Thompson 69], an unrelated language created by Thompson during the Multics days. Bon in turn was named either after his wife Bonnie, or (according to an encyclopedia quotation in its manual), after a religion whose rituals involve the murmuring of magic formulas.

Origins: the languages

BCPL was designed by Martin Richards in the mid-1960s while he was visiting MIT, and was used during the early 1970s for several interesting projects, among them the OS6 operating system at Oxford [Stoy 72], and parts of the seminal Alto work at Xerox PARC [Thacker 79]. We became familiar with it because the MIT CTSS system [Corbato 62] on which Richards worked was used for Multics development. The original BCPL compiler was transported both to Multics and to the GE-635 GECOS system by Rudd Canaday and others at Bell Labs [Canaday 69]; during the final throes of Multics's life at Bell Labs and immediately after, it was the language of choice among the group of people who would later become involved with Unix.

BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and

Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are 'close to the machine' in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. With less success, they also use library procedures to specify interesting control constructs such as coroutines and procedure closures. At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.

BCPL, B and C differ syntactically in many details, but broadly they are similar. Programs consist of a sequence of global declarations and function (procedure) declarations. Procedures can be nested in BCPL, but may not refer to non-static objects defined in containing procedures. B and C avoid this restriction by imposing a more severe one: no nested procedures at all. Each of the languages (except for earliest versions of B) recognizes separate compilation, and provides a means for including text from named files.

Several syntactic and lexical mechanisms of BCPL are more elegant and regular than those of B and C. For example, BCPL's procedure and data declarations have a more uniform structure, and it supplies a more complete set of looping constructs. Although BCPL programs are notionally supplied from an undelimited stream of characters, clever rules allow most semicolons to be elided after statements that end on a line boundary. B and C omit this convenience, and end most statements with semicolons. In spite of the differences, most of the statements and operators of BCPL map directly into corresponding B and C.

Some of the structural differences between BCPL and B stemmed from limitations on intermediate memory. For example, BCPL declarations may take the form

```
let P1 be command
and P2 be command
and P3 be command
 ...
```

where the program text represented by the commands contains whole procedures. The subdeclarations connected by `and` occur simultaneously, so the name `P3` is known inside procedure `P1`. Similarly, BCPL can package a group of declarations and statements into an expression that yields a value, for example

```
E1 := valof $( declarations ; commands ; resultis E2 $) + 1
```

The BCPL compiler readily handled such constructs by storing and analyzing a parsed representation of the entire program in memory before producing output. Storage limitations on the B compiler demanded a one-pass technique in which output was generated as soon as possible, and the syntactic redesign that made this possible was carried forward into C.

Certain less pleasant aspects of BCPL owed to its own technological problems and were consciously avoided in the design of B. For example, BCPL uses a 'global vector' mechanism for communicating between separately compiled programs. In this scheme, the programmer explicitly associates the name of each externally visible procedure and data object with a numeric offset in the global vector; the linkage is accomplished in the compiled code by using these numeric offsets. B evaded this inconvenience initially by insisting that the entire program be presented all at once to the compiler. Later implementations of B, and all those of C, use a conventional linker to resolve external names occurring in files compiled separately, instead of placing the burden of assigning offsets on the programmer.

Other fiddles in the transition from BCPL to B were introduced as a matter of taste, and some remain controversial, for example the decision to use the single character = for assignment instead of `:=`. Similarly, B uses `/* */` to enclose comments, where BCPL uses `//`, to ignore text up to the end of the line. The legacy of PL/I is evident here. (C++ has resurrected the BCPL comment convention.) Fortran influenced the syntax of declarations: B declarations begin with a specifier like `auto` or `static`, followed by a list of names, and C not only followed this style but ornamented it by placing its type keywords at the start of declarations.

Not every difference between the BCPL language documented in Richards's book [Richards 79] and B was deliberate; we started from an earlier version of BCPL [Richards 67]. For example, the `endcase` that escapes from a BCPL `switchon` statement was not present in the language when we learned it in the 1960s, and so the overloading of the `break` keyword to escape from the B and C `switch` statement owes to divergent evolution rather than conscious change.

In contrast to the pervasive syntax variation that occurred during the creation of B, the core semantic content of BCPL its type structure and expression evaluation rules remained intact. Both languages are typeless, or rather have a single data type, the 'word,' or 'cell,' a fixed-length bit pattern. Memory in these languages consists of a linear array of such cells, and the meaning of the contents of a cell depends on the operation applied. The + operator, for example, simply adds its operands using the machine's integer add instruction, and the other arithmetic operations are equally unconscious of the actual meaning of their operands. Because memory is a linear array, it is possible to interpret the value in a cell as an index in this array, and BCPL supplies an operator for this purpose. In the original language it was spelled `rv`, and later `!`, while B uses the unary `*`. Thus, if `p` is a cell containing the index of (or address of, or pointer to) another cell, `*p` refers to the contents of the pointed-to cell, either as a value in an expression or as the target of an assignment.

Because pointers in BCPL and B are merely integer indices in the memory array, arithmetic on them is meaningful: if `p` is the address of a cell, then `p+1` is the address of the next cell. This convention is the basis for the semantics of arrays in both languages. When in BCPL one writes

```
let V = vec 10
```

or in B,

```
auto V[10];
```

the effect is the same: a cell named `V` is allocated, then another group of 10 contiguous cells is set aside, and the memory index of the first of these is placed into `V`. By a general rule, in B the expression

```
*(V+i)
```

adds `V` and `i`, and refers to the `i`-th location after `V`. Both BCPL and B each add special notation to sweeten such array accesses; in B an equivalent expression is

```
V[i]
```

and in BCPL

```
V!i
```

This approach to arrays was unusual even at the time; C would later assimilate it in an even less conventional way.

None of BCPL, B, or C supports character data strongly in the language; each treats strings much like vectors of integers and supplements general rules by a few conventions. In both BCPL and B a string literal denotes the address of a static area initialized with the characters of the string, packed into cells. In BCPL, the first packed byte contains the number of characters in the string; in B, there is no count and strings are terminated by a special character, which B spelled '`*e`'. This change was made partially to avoid the limitation on the length of a string caused by holding the count in an 8- or 9-bit slot, and partly because maintaining the count seemed, in our experience, less convenient than using a terminator.

Individual characters in a BCPL string were usually manipulated by spreading the string out into another array, one character per cell, and then repacking it later; B provided corresponding routines, but people more often used other library functions that accessed or replaced individual characters in a string.

More History

After the TMG version of B was working, Thompson rewrote B in itself (a bootstrapping step). During development, he continually struggled against memory limitations: each language addition inflated the compiler so it could barely fit, but each rewrite taking advantage of the feature reduced its size. For example, B introduced generalized assignment operators, using `x=+y` to add `y` to `x`. The notation came from Algol 68 [Wijngaarden 75] via McIlroy, who had incorporated it into his version of TMG. (In B and early C, the operator was spelled =+ instead of += ; this mistake, repaired in 1976, was induced by a seductively easy way of handling the first form in B's lexical analyzer.)

Thompson went a step further by inventing the ++ and −− operators, which increment or decrement; their prefix or postfix position determines whether the alteration occurs before or after noting the value of the operand. They were not in the earliest versions of B, but appeared along the way. People often guess that they were created to use the auto-increment and auto-decrement address modes provided by the DEC PDP-11 on which C and Unix first became popular. This is historically impossible, since there was no PDP-11 when B was developed. The PDP-7, however, did have a few 'auto-increment' memory cells, with the property that an indirect memory reference through them incremented the cell. This feature probably suggested such operators to Thompson; the generalization to make them both prefix and postfix was his own. Indeed, the auto-increment cells were not used directly in implementation of the operators, and a stronger motivation for the innovation was probably his observation that the translation of `++x` was smaller than that of `x=x+1`.

The B compiler on the PDP-7 did not generate machine instructions, but instead 'threaded code' [Bell 72], an interpretive scheme in which the compiler's output consists of a sequence of addresses of code fragments that perform the elementary operations. The operations typically   in particular for B   act on a simple stack machine.

On the PDP-7 Unix system, only a few things were written in B except B itself, because the machine was too small and too slow to do more than experiment; rewriting the operating system and the utilities wholly into B was too expensive a step to seem feasible. At some point Thompson relieved the address-space crunch by offering a 'virtual B' compiler that allowed the interpreted program to occupy more than 8K bytes by paging the code and data within the interpreter, but it was too slow to be practical for the common utilities. Still, some utilities written in B appeared, including an early version of the variable-precision calculator *dc* familiar to Unix users [McIlroy 79]. The most ambitious enterprise I undertook was a genuine cross-compiler that translated B to GE-635 machine instructions, not threaded code. It was a small *tour de force*: a full B compiler, written in its own language and generating code for a 36-bit mainframe, that ran on an 18-bit machine with 4K words of user address space. This project was possible only because of the simplicity of the B language and its run-time system.

Although we entertained occasional thoughts about implementing one of the major languages of the time like Fortran, PL/I, or Algol 68, such a project seemed hopelessly large for our resources: much simpler and smaller tools were called for. All these languages influenced our work, but it was more fun to do things on our own.

By 1970, the Unix project had shown enough promise that we were able to acquire the new DEC PDP-11. The processor was among the first of its line delivered by DEC, and three months passed before its disk arrived. Making B programs run on it using the threaded technique required only writing the code fragments for the operators, and a simple assembler which I coded in B; soon, *dc* became the first interesting program to be tested, before any operating system, on our PDP-11. Almost as rapidly, still waiting for the disk, Thompson recoded the Unix kernel and some basic commands in PDP-11 assembly language. Of the 24K bytes of memory on the machine, the earliest PDP-11 Unix system used 12K bytes for the operating system, a tiny space for user programs, and the remainder as a RAM disk. This version was only for testing, not for real work; the machine marked time by enumerating closed knight's tours on chess boards of various sizes. Once its disk appeared, we quickly migrated to it after transliterating assembly-language commands to the PDP-11 dialect, and porting those already in B.

By 1971, our miniature computer center was beginning to have users. We all wanted to create interesting software more easily. Using assembler was dreary enough that B, despite its performance problems, had been supplemented by a small library of useful service routines and was being used for more and more new programs. Among the more notable results of this period was Steve Johnson's first version of the *yacc* parser-generator [Johnson 79a].

The Problems of B

The machines on which we first used BCPL and then B were word-addressed, and these languages' single data type, the 'cell,' comfortably equated with the hardware machine word. The advent of the PDP-11 exposed several inadequacies of B's semantic model. First, its character-handling mechanisms, inherited with few changes from BCPL, were clumsy: using library procedures to spread packed strings into individual cells and then repack, or to access and replace individual characters, began to feel awkward, even silly, on a byte-oriented machine.

Second, although the original PDP-11 did not provide for floating-point arithmetic, the manufacturer promised that it would soon be available. Floating-point operations had been added to BCPL in our Multics and GCOS compilers by defining special operators, but the mechanism was possible only because on the relevant machines, a single word was large enough to contain a floating-point number; this was not true on the 16-bit PDP-11.

Finally, the B and BCPL model implied overhead in dealing with pointers: the language rules, by defining a pointer as an index in an array of words, forced pointers to be represented as word indices. Each pointer reference generated a run-time scale conversion from the pointer to the byte address expected by the hardware.

For all these reasons, it seemed that a typing scheme was necessary to cope with characters and byte addressing, and to prepare for the coming floating-point hardware. Other issues, particularly type safety and interface checking, did not seem as important then as they became later.

Aside from the problems with the language itself, the B compiler's threaded-code technique yielded programs so much slower than their assembly-language counterparts that we discounted the possibility of recoding the operating system or its central utilities in B.

In 1971 I began to extend the B language by adding a character type and also rewrote its compiler to generate PDP-11 machine instructions instead of threaded code. Thus the transition from B to C was contemporaneous with the creation of a compiler capable of producing programs fast and small enough to compete with assembly language. I called the slightly-extended language NB, for 'new B.'

Embryonic C

NB existed so briefly that no full description of it was written. It supplied the types `int` and `char`, arrays of them, and pointers to them, declared in a style typified by

```
int i, j;
char c, d;
int iarray[10];
int ipointer[];
char carray[10];
char cpointer[];
```

The semantics of arrays remained exactly as in B and BCPL: the declarations of `iarray` and `carray` create cells dynamically initialized with a value pointing to the first of a sequence of 10 integers and characters respectively. The declarations for `ipointer` and `cpointer` omit the size, to assert that no storage should be allocated automatically. Within procedures, the language's interpretation of the pointers was identical to that of the array variables: a pointer declaration created a cell differing from an array declaration only in that the programmer was expected to assign a referent, instead of letting the compiler allocate the space and initialize the cell.

Values stored in the cells bound to array and pointer names were the machine addresses, measured in bytes, of the corresponding storage area. Therefore, indirection through a pointer implied no run-time overhead to scale the pointer from word to byte offset. On the other hand, the machine code for array subscripting and pointer arithmetic now depended on the type of the array or the pointer: to compute `iarray[i]` or `ipointer+i` implied scaling the addend `i` by the size of the object referred to.

These semantics represented an easy transition from B, and I experimented with them for some months. Problems became evident when I tried to extend the type notation, especially to add structured (record) types. Structures, it seemed, should map in an intuitive way onto memory in the machine, but in a structure containing an array, there was no good place to stash the pointer containing the base of the array, nor any convenient way to arrange that it be initialized. For example, the directory entries of early Unix systems might be described in C as

```
struct {
        int     inumber;
        char    name[14];
};
```

I wanted the structure not merely to characterize an abstract object but also to describe a collection of bits that might be read from a directory. Where could the compiler hide the pointer to `name` that the semantics demanded? Even if structures were thought of more abstractly, and the space for pointers could be hidden somehow, how could I handle the technical problem of properly initializing these pointers when allocating a complicated object, perhaps one that specified structures containing arrays containing structures to arbitrary depth?

The solution constituted the crucial jump in the evolutionary chain between typeless BCPL and typed C. It eliminated the materialization of the pointer in storage, and instead caused the creation of the pointer when the array name is mentioned in an expression. The rule, which survives in today's C, is that values of array type are converted, when they appear in expressions, into pointers to the first of the objects making up the array.

This invention enabled most existing B code to continue to work, despite the underlying shift in the language's semantics. The few programs that assigned new values to an array name to adjust its origin   possible in B and BCPL, meaningless in C   were easily repaired. More important, the new language retained a coherent and workable (if unusual) explanation of the semantics of arrays, while opening the way to a more comprehensive type structure.

The second innovation that most clearly distinguishes C from its predecessors is this fuller type structure and especially its expression in the syntax of declarations. NB offered the basic types `int` and `char`, together with arrays of them, and pointers to them, but no further ways of composition. Generalization was required: given an object of any type, it should be possible to describe a new object that gathers several into an array, yields it from a function, or is a pointer to it.

For each object of such a composed type, there was already a way to mention the underlying object: index the array, call the function, use the indirection operator on the pointer. Analogical reasoning led to a declaration syntax for names mirroring that of the expression syntax in which the names typically appear. Thus,

```
int i, *pi, **ppi;
```

declare an integer, a pointer to an integer, a pointer to a pointer to an integer. The syntax of these declarations reflects the observation that `i`, `*pi`, and `**ppi` all yield an `int` type when used in an expression. Similarly,

```
int f(), *f(), (*f)();
```

declare a function returning an integer, a function returning a pointer to an integer, a pointer to a function returning an integer;

```
int *api[10], (*pai)[10];
```

declare an array of pointers to integers, and a pointer to an array of integers. In all these cases the declaration of a variable resembles its usage in an expression whose type is the one named at the head of the declaration.

The scheme of type composition adopted by C owes considerable debt to Algol 68, although it did not, perhaps, emerge in a form that Algol's adherents would approve of. The central notion I captured from Algol was a type structure based on atomic types (including structures), composed into arrays, pointers (references), and functions (procedures). Algol 68's concept of unions and casts also had an influence that appeared later.

After creating the type system, the associated syntax, and the compiler for the new language, I felt that it deserved a new name; NB seemed insufficiently distinctive. I decided to follow the single-letter style and called it C, leaving open the question whether the name represented a progression through the alphabet or through the letters in BCPL.

Neonatal C

Rapid changes continued after the language had been named, for example the introduction of the `&&` and `||` operators. In BCPL and B, the evaluation of expressions depends on context: within `if` and other conditional statements that compare an expression's value with zero, these languages place a special interpretation on the `and` (`&`) and `or` (`|`) operators. In ordinary contexts, they operate bitwise, but in the B statement

```
if (e1 & e2) ...
```

the compiler must evaluate `e1` and if it is non-zero, evaluate `e2`, and if it too is non-zero, elaborate the statement dependent on the `if`. The requirement descends recursively on `&` and `|` operators within `e1` and `e2`. The short-circuit semantics of the Boolean operators in such 'truth-value' context seemed desirable, but the overloading of the operators was difficult to explain and use. At the suggestion of Alan Snyder, I introduced the `&&` and `||` operators to make the mechanism more explicit.

Their tardy introduction explains an infelicity of C's precedence rules. In B one writes

```
if (a==b & c) ...
```

to check whether `a` equals `b` and `c` is non-zero; in such a conditional expression it is better that `&` have lower precedence than ==. In converting from B to C, one wants to replace `&` by `&&` in such a statement; to make the conversion less painful, we decided to keep the precedence of the `&` operator the same relative to ==, and merely split the precedence of `&&` slightly from `&`. Today, it seems that it would have been preferable to move the relative precedences of `&` and ==, and thereby simplify a common C idiom: to test a masked value against another value, one must write

```
if ((a&mask) == b) ...
```

where the inner parentheses are required but easily forgotten.

Many other changes occurred around 1972-3, but the most important was the introduction of the preprocessor, partly at the urging of Alan Snyder [Snyder 74], but also in recognition of the utility of the the file-inclusion mechanisms available in BCPL and PL/I. Its original version was exceedingly simple, and provided only included files and simple string replacements: `#include` and `#define` of parameterless macros. Soon thereafter, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation. The preprocessor was originally considered an optional adjunct to the language itself. Indeed, for some years, it was not even invoked unless the source program contained a special signal at its beginning. This attitude persisted, and explains both the incomplete integration of the syntax of the preprocessor with the rest of the language and the imprecision of its description in early reference manuals.

Portability

By early 1973, the essentials of modern C were complete. The language and compiler were strong enough to permit us to rewrite the Unix kernel for the PDP-11 in C during the summer of that year. (Thompson had made a brief attempt to produce a system coded in an early version of C before structures in 1972, but gave up the effort.) Also during this period, the compiler was retargeted to other nearby machines, particularly the Honeywell 635 and IBM 360/370; because the language could not live in isolation, the prototypes for the modern libraries were developed. In particular, Lesk wrote a 'portable I/O package' [Lesk 72] that was later reworked to become the C 'standard I/O' routines. In 1978 Brian Kernighan and I published *The C Programming Language* [Kernighan 78]. Although it did not describe some additions that soon became common, this book served as the language reference until a formal standard was adopted more than ten years later. Although we worked closely together on this book, there was a clear division of labor: Kernighan wrote almost all the expository material, while I was responsible for the appendix containing the reference manual and the chapter on interfacing with the Unix system.

During 1973-1980, the language grew a bit: the type structure gained unsigned, long, union, and enumeration types, and structures became nearly first-class objects (lacking only a notation for literals). Equally important developments appeared in its environment and the accompanying technology. Writing the Unix kernel in C had given us enough confidence in the language's usefulness and efficiency that we began to recode the system's utilities and tools as well, and then to move the most interesting among them to the other platforms. As described in [Johnson 78a], we discovered that the hardest problems in propagating Unix tools lay not in the interaction of the C language with new hardware, but in adapting to the existing software of other operating systems. Thus Steve Johnson began to work on *pcc*, a C compiler intended to be easy to retarget to new machines [Johnson 78b], while he, Thompson, and I began to move the Unix system itself to the Interdata 8/32 computer.

The language changes during this period, especially around 1977, were largely focused on considerations of portability and type safety, in an effort to cope with the problems we foresaw and observed in moving a considerable body of code to the new Interdata platform. C at that time still manifested strong signs of its typeless origins. Pointers, for example, were barely distinguished from integral memory indices in early language manuals or extant code; the similarity of the arithmetic properties of character pointers and unsigned integers made it hard to resist the temptation to identify them. The `unsigned` types were added to make unsigned arithmetic available without confusing it with pointer manipulation. Similarly, the early language condoned assignments between integers and pointers, but this practice began to be discouraged; a notation for type conversions (called 'casts' from the example of Algol 68) was invented to specify type conversions more explicitly. Beguiled by the example of PL/I, early C did not tie structure pointers firmly to the structures they pointed to, and permitted programmers to write `pointer->member` almost without regard to the type of `pointer`; such an expression was taken uncritically as a reference to a region of memory designated by the pointer, while the member name specified only an offset and a type.

Although the first edition of K&R described most of the rules that brought C's type structure to its present form, many programs written in the older, more relaxed style persisted, and so did compilers that tolerated it. To encourage people to pay more attention to the official language rules, to detect legal but suspicious constructions, and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his *pcc* compiler to produce *lint* [Johnson 79b], which scanned a set of files and remarked on dubious constructions.

Growth in Usage

The success of our portability experiment on the Interdata 8/32 soon led to another by Tom London and John Reiser on the DEC VAX 11/780. This machine became much more popular than the Interdata, and Unix and the C language began to spread rapidly, both within AT&T and outside. Although by the middle 1970s Unix was in use by a variety of projects within the Bell

System as well as a small group of research-oriented industrial, academic, and government organizations outside our company, its real growth began only after portability had been achieved. Of particular note were the System III and System V versions of the system from the emerging Computer Systems division of AT&T, based on work by the company's development and research groups, and the BSD series of releases by the University of California at Berkeley that derived from research organizations in Bell Laboratories.

During the 1980s the use of the C language spread widely, and compilers became available on nearly every machine architecture and operating system; in particular it became popular as a programming tool for personal computers, both for manufacturers of commercial software for these machines, and for end-users interesting in programming. At the start of the decade, nearly every compiler was based on Johnson's *pcc*; by 1985 there were many independently-produced compiler products.

Standardization

By 1982 it was clear that C needed formal standardization. The best approximation to a standard, the first edition of K&R, no longer described the language in actual use; in particular, it mentioned neither the `void` or `enum` types. While it foreshadowed the newer approach to structures, only after it was published did the language support assigning them, passing them to and from functions, and associating the names of members firmly with the structure or union containing them. Although compilers distributed by AT&T incorporated these changes, and most of the purveyors of compilers not based on *pcc* quickly picked up them up, there remained no complete, authoritative description of the language.

The first edition of K&R was also insufficiently precise on many details of the language, and it became increasingly impractical to regard *pcc* as a 'reference compiler;' it did not perfectly embody even the language described by K&R, let alone subsequent extensions. Finally, the incipient use of C in projects subject to commercial and government contract meant that the imprimatur of an official standard was important. Thus (at the urging of M. D. McIlroy), ANSI established the X3J11 committee under the direction of CBEMA in the summer of 1983, with the goal of producing a C standard. X3J11 produced its report [ANSI 89] at the end of 1989, and subsequently this standard was accepted by ISO as ISO/IEC 9899-1990.

From the beginning, the X3J11 committee took a cautious, conservative view of language extensions. Much to my satisfaction, they took seriously their goal: 'to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.' [ANSI 89] The committee realized that mere promulgation of a standard does not make the world change.

X3J11 introduced only one genuinely important change to the language itself: it incorporated the types of formal arguments in the type signature of a function, using syntax borrowed from C++ [Stroustrup 86]. In the old style, external functions were declared like this:

```
double sin();
```

which says only that `sin` is a function returning a `double` (that is, double-precision floating-point) value. In the new style, this better rendered

```
double sin(double);
```

to make the argument type explicit and thus encourage better type checking and appropriate conversion. Even this addition, though it produced a noticeably better language, caused difficulties. The committee justifiably felt that simply outlawing 'old-style' function definitions and declarations was not feasible, yet also agreed that the new forms were better. The inevitable compromise was as good as it could have been, though the language definition is complicated by permitting both forms, and writers of portable software must contend with compilers not yet brought up to standard.

X3J11 also introduced a host of smaller additions and adjustments, for example, the type qualifiers `const` and `volatile`, and slightly different type promotion rules. Nevertheless, the standardization process did not change the character of the language. In particular, the C standard did not attempt to specify formally the language semantics, and so there can be dispute over fine points; nevertheless, it successfully accounted for changes in usage since the original description, and is sufficiently precise to base implementations on it.

Thus the core C language escaped nearly unscathed from the standardization process, and the Standard emerged more as a better, careful codification than a new invention. More important changes took place in the language's surroundings: the preprocessor and the library. The preprocessor performs macro substitution, using conventions distinct from the rest of the language. Its interaction with the compiler had never been well-described, and X3J11 attempted to remedy the situation. The result is noticeably better than the explanation in the first edition of K&R; besides being more comprehensive, it provides operations, like token concatenation, previously available only by accidents of implementation.

X3J11 correctly believed that a full and careful description of a standard C library was as important as its work on the language itself. The C language itself does not provide for input-output or any other interaction with the outside world, and thus depends on a set of standard procedures. At the time of publication of K&R, C was thought of mainly as the system programming language of Unix; although we provided examples of library routines intended to be readily transportable to other operating systems, underlying support from Unix was implicitly understood. Thus, the X3J11 committee spent much of its time designing and documenting a set of library routines required to be available in all conforming implementations.

By the rules of the standards process, the current activity of the X3J11 committee is confined to issuing interpretations on the existing standard. However, an informal group originally convened by Rex Jaeschke as NCEG (Numerical C Extensions Group) has been officially accepted as subgroup X3J11.1, and they continue to consider extensions to C. As the name implies, many of these possible extensions are intended to make the language more suitable for numerical use: for example, multi-dimensional arrays whose bounds are dynamically determined, incorporation of facilities for dealing with IEEE arithmetic, and making the language more effective on machines with vector or other advanced architectural features. Not all the possible extensions are specifically numerical; they include a notation for structure literals.

Successors

C and even B have several direct descendants, though they do not rival Pascal in generating progeny. One side branch developed early. When Steve Johnson visited the University of Waterloo on sabbatical in 1972, he brought B with him. It became popular on the Honeywell machines there, and later spawned Eh and Zed (the Canadian answers to 'what follows B?'). When Johnson returned to Bell Labs in 1973, he was disconcerted to find that the language whose seeds he brought to Canada had evolved back home; even his own *yacc* program had been rewritten in C, by Alan Snyder.

More recent descendants of C proper include Concurrent C [Gehani 89], Objective C [Cox 86], C* [Thinking 90], and especially C++ [Stroustrup 86]. The language is also widely used as an intermediate representation (essentially, as a portable assembly language) for a wide variety of compilers, both for direct descendents like C++, and independent languages like Modula 3 [Nelson 91] and Eiffel [Meyer 88].

Critique

Two ideas are most characteristic of C among languages of its class: the relationship between arrays and pointers, and the way in which declaration syntax mimics expression syntax. They are also among its most frequently criticized features, and often serve as stumbling blocks to the beginner. In both cases, historical accidents or mistakes have exacerbated their difficulty. The most important of these has been the tolerance of C compilers to errors in type. As should be clear from the history above, C evolved from typeless languages. It did not suddenly appear to its earliest users and developers as an entirely new language with its own rules; instead we continually had to adapt existing programs as the language developed, and make allowance for an existing body of code. (Later, the ANSI X3J11 committee standardizing C would face the same problem.)

Compilers in 1977, and even well after, did not complain about usages such as assigning between integers and pointers or using objects of the wrong type to refer to structure members. Although the language definition presented in the first edition of K&R was reasonably (though not completely) coherent in its treatment of type rules, that book admitted that existing compilers didn't enforce them. Moreover, some rules designed to ease early transitions contributed to later confusion. For example, the empty square brackets in the function declaration

```
int f(a) int a[]; { ... }
```

are a living fossil, a remnant of NB's way of declaring a pointer; `a` is, in this special case only, interpreted in C as a pointer. The notation survived in part for the sake of compatibility, in part under the rationalization that it would allow programmers to communicate to their readers an intent to pass `f` a pointer generated from an array, rather than a reference to a single integer. Unfortunately, it serves as much to confuse the learner as to alert the reader.

In K&R C, supplying arguments of the proper type to a function call was the responsibility of the programmer, and the extant compilers did not check for type agreement. The failure of the original language to include argument types in the type signature of a function was a significant weakness, indeed the one that required the X3J11 committee's boldest and most painful innovation to repair. The early design is explained (if not justified) by my avoidance of technological problems, especially cross-checking between separately-compiled source files, and my incomplete assimilation of the implications of moving between an untyped to a typed language. The *lint* program, mentioned above, tried to alleviate the problem: among its other functions, *lint* checks the consistency and coherency of a whole program by scanning a set of source files, comparing the types of function arguments used in calls with those in their definitions.

An accident of syntax contributed to the perceived complexity of the language. The indirection operator, spelled `*` in C, is syntactically a unary prefix operator, just as in BCPL and B. This works well in simple expressions, but in more complex cases, parentheses are required to direct the parsing. For example, to distinguish indirection through the value returned by a function from calling a function designated by a pointer, one writes `*fp()` and `(*pf)()` respectively. The style used in expressions carries through to declarations, so the names might be declared

```
int *fp();
int (*pf)();
```

In more ornate but still realistic cases, things become worse:

```
int *(*pfp)();
```

is a pointer to a function returning a pointer to an integer. There are two effects occurring. Most important, C has a relatively rich set of ways of describing types (compared, say, with Pascal). Declarations in languages as expressive as C Algol 68, for example describe objects equally hard to understand, simply because the objects themselves are complex. A second effect owes to details of the syntax. Declarations in C must be read in an 'inside-out' style that many find difficult to grasp [Anderson 80]. Sethi [Sethi 81] observed that many of the nested declarations and expressions would become simpler if the indirection operator had been taken as a postfix operator instead of prefix, but by then it was too late to change.

In spite of its difficulties, I believe that the C's approach to declarations remains plausible, and am comfortable with it; it is a useful unifying principle.

The other characteristic feature of C, its treatment of arrays, is more suspect on practical grounds, though it also has real virtues. Although the relationship between pointers and arrays is unusual, it can be learned. Moreover, the language shows considerable power to describe important concepts, for example, vectors whose length varies at run time, with only a few basic rules and conventions. In particular, character strings are handled by the same mechanisms as any other array, plus the convention that a null character terminates a string. It is interesting to compare C's approach with that of two nearly contemporaneous languages, Algol 68 and Pascal [Jensen 74]. Arrays in Algol 68 either have fixed bounds, or are 'flexible:' considerable mechanism is required both in the language definition, and in compilers, to accommodate flexible arrays (and not all compilers fully implement them.) Original Pascal had only fixed-sized arrays and strings, and this proved confining [Kernighan 81]. Later, this was partially fixed, though the resulting language is not yet universally available.

C treats strings as arrays of characters conventionally terminated by a marker. Aside from one special rule about initialization by string literals, the semantics of strings are fully subsumed by more general rules governing all arrays, and as a result the language is simpler to describe and to translate than one incorporating the string as a unique data type. Some costs accrue from its approach: certain string operations are more expensive than in other designs because application code or a library routine must occasionally search for the end of a string, because few built-in operations are available, and because the burden of storage management for strings falls more heavily on the user. Nevertheless, C's approach to strings works well.

On the other hand, C's treatment of arrays in general (not just strings) has unfortunate implications both for optimization and for future extensions. The prevalence of pointers in C programs, whether those declared explicitly or arising from arrays, means that optimizers must be cautious, and must use careful dataflow techniques to achieve good results. Sophisticated compilers can understand what most pointers can possibly change, but some important usages remain difficult to analyze. For example, functions with pointer arguments derived from arrays are hard to compile into efficient code on vector machines, because it is seldom possible to determine that one argument pointer does not overlap data also referred to by another argument, or accessible externally. More fundamentally, the definition of C so specifically describes the semantics of arrays that changes or extensions treating arrays as more primitive objects, and permitting operations on them as wholes, become hard to fit into the existing language. Even extensions to permit the declaration and use of multidimensional arrays whose size is determined dynamically are not entirely straightforward [MacDonald 89] [Ritchie 90], although they would make it much easier to write numerical libraries in C. Thus, C covers the most important uses of strings and arrays arising in practice by a uniform and simple mechanism, but leaves problems for highly efficient implementations and for extensions.

Many smaller infelicities exist in the language and its description besides those discussed above, of course. There are also general criticisms to be lodged that transcend detailed points. Chief among these is that the language and its generally-expected environment provide little help for writing very large systems. The naming structure provides only two main levels, 'external' (visible everywhere) and 'internal' (within a single procedure). An intermediate level of visibility (within a single file of data and procedures) is weakly tied to the language definition. Thus, there is little direct support for modularization, and project designers are forced to create their own conventions.

Similarly, C itself provides two durations of storage: 'automatic' objects that exist while control resides in or below a procedure, and 'static,' existing throughout execution of a program. Off-stack, dynamically-allocated storage is provided only by a library routine and the burden of managing it is placed on the programmer: C is hostile to automatic garbage collection.

Whence Success?

C has become successful to an extent far surpassing any early expectations. What qualities contributed to its widespread use?

Doubtless the success of Unix itself was the most important factor; it made the language available to hundreds of thousands of people. Conversely, of course, Unix's use of C and its consequent portability to a wide variety of machines was important in the system's success. But the language's invasion of other environments suggests more fundamental merits.

Despite some aspects mysterious to the beginner and occasionally even to the adept, C remains a simple and small language, translatable with simple and small compilers. Its types and operations are well-grounded in those provided by real machines, and for people used to how computers work, learning the idioms for generating time- and space-efficient programs is not difficult. At the same time the language is sufficiently abstracted from machine details that program portability can be achieved.

Equally important, C and its central library support always remained in touch with a real environment. It was not designed in isolation to prove a point, or to serve as an example, but as a tool to write programs that did useful things; it was always meant to interact with a larger operating system, and was regarded as a tool to build larger tools. A parsimonious, pragmatic approach influenced the things that went into C: it covers the essential needs of many programmers, but does not try to supply too much.

Finally, despite the changes that it has undergone since its first published description, which was admittedly informal and incomplete, the actual C language as seen by millions of users using many different compilers has remained remarkably stable and unified compared to those of similarly widespread currency, for example Pascal and Fortran. There are differing dialects of C most noticeably, those described by the older K&R and the newer Standard C but on the whole, C has remained freer of proprietary extensions than other languages. Perhaps the most significant extensions are the ‘far’ and ‘near’ pointer qualifications intended to deal with peculiarities of some Intel processors. Although C was not originally designed with portability as a prime goal, it succeeded in expressing programs, even including operating systems, on machines ranging from the smallest personal computers through the mightiest supercomputers.

C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.

Acknowledgments

It is worth summarizing compactly the roles of the direct contributors to today’s C language. Ken Thompson created the B language in 1969-70; it was derived directly from Martin Richards’s BCPL. Dennis Ritchie turned B into C during 1971-73, keeping most of B’s syntax while adding types and many other changes, and writing the first compiler. Ritchie, Alan Snyder, Steven C. Johnson, Michael Lesk, and Thompson contributed language ideas during 1972-1977, and Johnson’s portable compiler remains widely used. During this period, the collection of library routines grew considerably, thanks to these people and many others at Bell Laboratories. In 1978, Brian Kernighan and Ritchie wrote the book that became the language definition for several years. Beginning in 1983, the ANSI X3J11 committee standardized the language. Especially notable in keeping its efforts on track were its officers Jim Brodie, Tom Plum, and P. J. Plauger, and the successive draft redactors, Larry Rosler and Dave Prosser.

I thank Brian Kernighan, Doug McIlroy, Dave Prosser, Peter Nelson, Rob Pike, Ken Thompson, and HOPL’s referees for advice in the preparation of this paper.

References

[ANSI 89] American National Standards Institute, *American National Standard for Information Systems Programming Language C,* X3.159-1989.

[Anderson 80] B. Anderson, ‘Type syntax in the language C: an object lesson in syntactic innovation,’ SIGPLAN Notices **15** (3), March, 1980, pp. 21-27.

[Bell 72] J. R. Bell, ‘Threaded Code,’ C. ACM **16** (6), pp. 370-372.

[Canaday 69] R. H. Canaday and D. M. Ritchie, ‘Bell Laboratories BCPL,’ AT&T Bell Laboratories internal memorandum, May, 1969.

[Corbato 62] F. J. Corbato, M. Merwin-Dagget, R. C. Daley, ‘An Experimental Time-sharing System,’ AFIPS Conf. Proc. SJCC, 1962, pp. 335-344.

[Cox 86] B. J. Cox and A. J. Novobilski, *Object-Oriented Programming: An Evolutionary*

Approach, Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.

[Gehani 89] N. H. Gehani and W. D. Roome, *Concurrent C,* Silicon Press: Summit, NJ, 1989.

[Jensen 74] K. Jensen and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag: New York, Heidelberg, Berlin. Second Edition, 1974.

[Johnson 73] S. C. Johnson and B. W. Kernighan, 'The Programming Language B,' Comp. Sci. Tech. Report #8, AT&T Bell Laboratories (January 1973).

[Johnson 78a] S. C. Johnson and D. M. Ritchie, 'Portability of C Programs and the UNIX System,' Bell Sys. Tech. J. **57** (6) (part 2), July-Aug, 1978.

[Johnson 78b] S. C. Johnson, 'A Portable Compiler: Theory and Practice,' Proc. 5th ACM POPL Symposium (January 1978).

[Johnson 79a] S. C. Johnson, 'Yet another compiler-compiler,' in *Unix Programmer's Manual,* Seventh Edition, Vol. 2A, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Johnson 79b] S. C. Johnson, 'Lint, a Program Checker,' in *Unix Programmer's Manual,* Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Kernighan 78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.

[Kernighan 81] B. W. Kernighan, 'Why Pascal is not my favorite programming language,' Comp. Sci. Tech. Rep. #100, AT&T Bell Laboratories, 1981.

[Lesk 73] M. E. Lesk, 'A Portable I/O Package,' AT&T Bell Laboratories internal memorandum ca. 1973.

[MacDonald 89] T. MacDonald, 'Arrays of variable length,' J. C Lang. Trans **1** (3), Dec. 1989, pp. 215-233.

[McClure 65] R. M. McClure, 'TMG A Syntax Directed Compiler,' Proc. 20th ACM National Conf. (1965), pp. 262-274.

[McIlroy 60] M. D. McIlroy, 'Macro Instruction Extensions of Compiler Languages,' C. ACM **3** (4), pp. 214-220.

[McIlroy 79] M. D. McIlroy and B. W. Kernighan, eds, *Unix Programmer's Manual,* Seventh Edition, Vol. I, AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Meyer 88] B. Meyer, *Object-oriented Software Construction,* Prentice-Hall: Englewood Cliffs, NJ, 1988.

[Nelson 91] G. Nelson, *Systems Programming with Modula-3,* Prentice-Hall: Englewood Cliffs, NJ, 1991.

[Organick 75] E. I. Organick, *The Multics System: An Examination of its Structure,* MIT Press: Cambridge, Mass., 1975.

[Richards 67] M. Richards, 'The BCPL Reference Manual,' MIT Project MAC Memorandum M-352, July 1967.

[Richards 79] M. Richards and C. Whitbey-Strevens, *BCPL: The Language and its Compiler,* Cambridge Univ. Press: Cambridge, 1979.

[Ritchie 78] D. M. Ritchie, 'UNIX: A Retrospective,' Bell Sys. Tech. J. **57** (6) (part 2), July-Aug, 1978.

[Ritchie 84] D. M. Ritchie, 'The Evolution of the UNIX Time-sharing System,' AT&T Bell Labs. Tech. J. **63** (8) (part 2), Oct. 1984.

[Ritchie 90] D. M. Ritchie, 'Variable-size arrays in C,' J. C Lang. Trans. **2** (2), Sept. 1990, pp. 81-86.

[Sethi 81] R. Sethi, 'Uniform syntax for type expressions and declarators,' Softw. Prac. and Exp. **11** (6), June 1981, pp. 623-628.

[Snyder 74] A. Snyder, *A Portable Compiler for the Language C,* MIT: Cambridge, Mass., 1974.

[Stoy 72] J. E. Stoy and C. Strachey, 'OS6 An experimental operating system for a small computer. Part I: General principles and structure,' Comp J. **15**, (Aug. 1972), pp.

117-124.

[Stroustrup 86] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.

[Thacker 79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. R. Boggs, 'Alto: A Personal Computer,' in *Computer Structures: Principles and Examples,* D. Sieworek, C. G. Bell, A. Newell, McGraw-Hill: New York, 1982.

[Thinking 90] *C* Programming Guide,* Thinking Machines Corp.: Cambridge Mass., 1990.

[Thompson 69] K. Thompson, 'Bon an Interactive Language,' undated AT&T Bell Laboratories internal memorandum (ca. 1969).

[Wijngaarden 75] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. Koster, M. Sintzoff, C. Lindsey, L. G. Meertens, R. G. Fisker, 'Revised report on the algorithmic language Algol 68,' Acta Informatica **5**, pp. 1-236.