A BYTE BOOK

LES HANCOCK · MORRIS KRIEGER

THE C PRIMER

The C Primer

Les Hancock

Morris Krieger

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland
Bogotá Hamburg Johannesburg London
Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging in Publication Data

Hancock, Les.
The C primer.

1. C (Computer program language) I. Krieger, Morris,
date. II. Title.
QA76.73.C15H36 1983 001.64'24 82-22897
ISBN 0-07-025981-X

1234567890 KGP/KGP 898765432

ISBN 0-07-025981-X

*This book was set in Times Roman and Monospace by the authors,
using a Graphic Systems Phototypesetter driven by
a PDP-11/45 running under the UNIX operating system.*

Printed and bound by The Kingsport Press.

Introduction

A primer is a book for beginners. This primer is intended for those programmers who, while they may know something about programming, know nothing whatever about the C language; and the amount of programming knowledge we do assume our readers have is minimal. We assume they have access to a computer that runs C, and that they know enough about programming to create source code files using a system editor and then compile and run those files.

We don't pretend this primer contains a complete description of the C language. The appendix lists the features of C we don't describe. Nor will we be discussing any topic that requires a deep knowledge of systems programming, which happens to be the kind of programming C is particularly intended for. The discussion of such concepts would certainly have made the book a hard chew for our intended readers. More importantly, it would have obscured the clean, stripped-to-essentials outline of C a beginner needs and should have.

Perhaps our most important omission is that we say nothing about file handling. Most programs, whatever language they're written in, read data from files and write data to files. As almost everyone knows, C was meant to run under the UNIX® operating system, which has an excellent set of file-handling facilities. But we say nothing about UNIX or any of the other operating systems on which C may now be run. Therefore, as you make your way through our primer, or after you've finished it, you will have to consult the user's manual that is supplied with your operating system and follow the procedures described there to discover how to read from and write to files on your machine.

®Bell Telephone Laboratories

While C is a clean, straightforward language, and this is particularly so at the beginner's level at which this primer is written, there are two things about C that a beginner with some knowledge of another high-level language may find bothersome. One has to do with C's relaxed attitude toward data typing; that is, the when, where, and how of declaring variables. In contrast to Pascal, for example, which insists that variables be declared before they can be used, and always used as declared, C does not invariably insist that variables be declared in advance. Instead, C has a system of automatic defaults. If a programmer neglects to declare a variable before using it, at its first use in a program C will assume the programmer meant the variable to be of whatever type seems to be sensible to the compiler under the circumstances. These default assumptions may not be what the programmer intended, and the result will be a great gnashing of teeth. To avoid confusion, the beginner needn't take advantage of C's relaxed attitude toward data typing if he doesn't want to. In fact, it would be better for him not to do so. Instead, he should explicitly declare all his variables.

Another source of potential difficulty for the beginner has to do with C's pointer notation. Pointers work by indirection, which is inherently a source of confusion. Experienced assembly language programmers will readily grasp what C pointers are and what they are meant to do. Others may not. The simplest solution for those who are not familiar with pointers and find them confusing is simply not to use them. In this case, their C programs will look and run much like the programs they might have written in one of the other structured languages. Hopefully, they will in time discover how useful C pointers can be.

During the planning of this primer, it became apparent to us that a set of exercises would be extremely useful to a beginner, but we put off preparing them. After we had completed the text, we discovered that someone had already written the exercises we had in mind. The exercises are published in the form of a series of puzzles under the title, *The C Puzzle Book*, by Alan R. Feuer.* We strongly urge our readers to obtain a copy of Feuer's book and work the appropriate puzzles as they complete the chapters in our primer. The fit between the two books is not perfect, but it is very good. In addition, Feuer touches on aspects of C we thought too advanced for a primer (e.g., casts, bitwise operators), but our readers should not find these topics difficult to understand.

Having completed this primer (and Feuer's puzzles), what can the

* Feuer, Alan R., *The C Puzzle Book*. Prentice-Hall, 1982.

interested beginner do next to consolidate and extend his knowledge of the C language? The answer is obvious: read *The C Programming Language*, by Kernighan and Ritchie.* Indeed, one of the objects of this primer is to bring the beginner up to the level where he can get more out of their book than he might otherwise for, truth to tell, *The C Programming Language* is not a book for beginning C programmers. It was in fact written to introduce C to experienced systems programmers for whom the underlying language and programming concepts were largely self-evident. The graceful clarity with which the book is written has in a sense proven deceptive, for it has made many beginners believe they could get through the book without too much trouble. Alas, they have instead found themselves hopelessly bogged down somewhere in Chapter 3. A warning to beginners is therefore still in order. Having completed this primer, they should find *The C Programming Language* much more accessible, but if they are unfamiliar with systems programming concepts there will still be much in Kernighan and Ritchie they will find difficult.

Another thing a beginner might do to learn what the programming art is all about is try working his way through the programs contained in *Software Tools*, by Kernighan and Plauger,† one of the best expositions on the art that has ever been written. We would recommend the original version of *Software Tools*, where the programs are written in Ratfor, for Ratfor and C are quite similar.

And, of course, the beginner should start writing his own programs in C as soon as possible. It is the only way to really learn the language.

Acknowledgements

First, we want to thank Ed Yourdon for allowing us to use the computing facilities of Yourdon, Inc. for the preparation of the text. This entire book was written and the final version typeset using Yourdon's PDP-11/45 and photocomposition equipment, all running under UNIX. We want also to thank the staff and computer operators at Yourdon, Inc. for their expert help and advice.

All or part of a draft of this book was read by Chris Terry, Mark Pearson, Thuy Nguyen, Tom Gibson, and Arden Phillips. We want to express our appreciation to them for the trouble they've taken. We are of course responsible for the contents of this book as it now exists.

* Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*. Prentice-Hall, 1978.

† Kernighan, Brian W. and P.J. Plauger, *Software Tools*. Addison-Wesley, 1976.

Table of Contents

Chapter 1
What C Is

C is a programming language developed at Bell Laboratories around 1972. It was designed and written by one man, Dennis Ritchie, who was then working closely with Ken Thompson on the UNIX operating system. UNIX was conceived as a sort of workshop full of tools for the software engineer, and C turned out to be the most basic tool of all. Nearly every software tool supplied with UNIX, including the operating system and the C compiler, is now written in C.

In the mid-1970s UNIX spread throughout Bell Labs. It was widely licensed to universities. Without any fuss, C began to replace the more familiar languages available on UNIX. No one pushed C. It wasn't made the "official" Bell Labs language. Seemingly self-propelled, without any advertisement, C's reputation spread and its pool of users grew. Ritchie seems to have been rather surprised that so many programmers preferred C to old standbys like Fortran or PL/I, or to new favorites like Pascal and APL. But that's what happened. By 1980 several C compilers were available from independent vendors, and C was running on various non-UNIX systems.

It's entirely in character for C to make such a modest debut. It belongs to a well-established family of languages whose tradition stresses low-key virtues: reliability, regularity, simplicity, ease of use. The members of this family are often called "structured" languages, since they're well suited to *structured programming*, a discipline intended to make programs easier to read and write. Structured programming became something of an ideology in the 1970s, and other languages hew to the party line more closely than C. The prize for purity is often given to Pascal, C's pretty sister. C wasn't meant to win prizes; it was meant to be friendly, capable, and reliable. Homely virtues these, but quite a few programmers who begin by falling in love with Pascal end up happily married to C.

C's direct ancestry is easy to trace. This is the line of descent:

Algol 60
Designed by an international committee, 1960

↓

CPL
(Combined Programming Language)
Cambridge and the University of London, 1963

↓

BCPL
(Basic Combined Programming Language)
Martin Richards, Cambridge, 1967

↓

B
Ken Thompson, Bell Labs, 1970

↓

C
Dennis Ritchie, Bell Labs, 1972

Though Algol appeared only a few years after Fortran, it's a much more sophisticated language, and for that reason has had enormous influence on programming language design. Its authors paid a great deal of attention to regularity of syntax, modular structure, and other features we tend to think of as "modern." Unfortunately, Algol never really caught on in the United States, probably because it seemed too abstract, too general. CPL was an attempt to bring Algol down to earth—in its inventors' words, to "retain contact . . . with the realities of an actual computer"*—a goal shared by C. Like Algol, CPL was big, with a host of features which did little to enhance its power but did make it hard to learn and difficult to implement. BCPL aimed to solve the problem by boiling CPL down to its basic good features. B, written by Ken Thompson for an early implementation of UNIX, is a further

* Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., Strachey, C., "The Main Features of CPL." *Computer Journal*, Vol. 6, 1963, p. 134.

simplification of CPL—and a very spare language it is indeed, though well suited for use on the hardware then available. But both BCPL and B carried economy of means so far that they became rather limited languages, useful only when dealing with certain kinds of problems. Ritchie's achievement in C was to restore some of this lost generality, mainly by the cunning use of data types. He managed to do this without sacrificing the simplicity or "computer contact" that were the design goals of CPL.

Like BCPL and B, C has the coherence that's often associated with one-man languages, other well-known examples being Lisp, Pascal, and APL. (Counterexamples include such many-headed monsters as PL/I, Algol 68, and Ada.) Following in his predecessors' small-but-beautiful footsteps, Ritchie was able to avoid the catastrophic complexity of languages that try to be all things to all men. Yet his minimalist approach didn't rob C of its power. By following a few simple, regular rules, C's limited stock of parts can be put together to make more complex parts, which can in turn be put together to form even more elaborate constructions. By way of comparison, think of the complex organic molecules that can be assembled from a dozen different atoms, or the symphonies that have been composed from the twelve notes of the chromatic scale. Simple building blocks (atoms and notes) are put together according to simple rules (of valency and harmony) to build more elaborate parts (radicals, chords) which are in turn used to create complex organisms and music of great beauty.

This ability to build complex programs out of simple elements is C's great strength. If C had a coat of arms, its motto might be *multum in parvo*: a lot from a little.

Languages written by one man usually reflect their author's field of expertise. Dennis Ritchie's field is systems software—computer languages, operating systems, program generators, text processors, etc.—and C is at its best when used to implement tools of this kind. Even though there's a good deal of generality built into C, let's be clear: it's not the language of choice for every application. You can, if you want, use C for writing everything from accounts receivable programs to video games: in principle, almost any computer language can do, one way or another, what any other language can do. And it is true that programs written in C run fast and take little storage space. But while an analysis of variance written in C may run faster than one written in APL, the APL program will be up and running first.

So C's special domain is systems software. Why is it so well suited to that field? Two reasons. First, it's a relatively low-level language that lets you specify every detail in a program's logic to achieve maximum computer efficiency. Second, it's a relatively high-level language that hides the details of the computer's architecture, thus promoting programming efficiency. The key to this paradox is the word *relative.* Relative to what? Or, to put it another way, what *is* C's place in the world of programming languages?

We can answer that question by referring to this hierarchy:

True dialogue

.

.

.

Artificial intelligence "dialogues"
Command languages (as in operating systems)
Problem-oriented languages
Machine-oriented languages
Assembly languages

.

.

.

Hardware

Reading from bottom to top, these categories go from the concrete to the abstract, from the highly detailed to the very general, from machine-oriented to human-oriented, and, more or less, from the past toward the future. The dots represent big leaps, with many steps left out. Early ancestors of the computer, like the Jacquard loom (1805) or Charles Babbage's "analytical engine" (1834), were programmed in hardware, and the day may come when we program a machine by having a chat with it, à la HAL 9000—but that certainly won't happen by the year 2001.

Assembly languages, which provide a fairly painless way for us to work directly with a computer's built-in instruction set, go back to the first days of electronic computers. Since they force you to think in terms of the hardware and to specify every operation in the machine's terms—move these bits into this register and add them to the bits in that other register, then place the result in memory at this location, and so on—they're very tedious to use, and errors are common. The early high-level languages, like Fortran and Algol, were created as alternatives to assembly languages. They were much more general, more abstract, allowing programmers to think in terms of the problem at hand rather than in terms of the computer's hardware. Logical structure could be visibly imposed on the program. It's the difference between writing `a = b + c` and writing

```
LHLD .c
PUSH H
POP B
LHLD .b
DAD B
SHLD .a
```

which is about the quickest way to say the same thing in the assembly language of the 8080 computer chip.

But the early software designers may have jumped too far up our hierarchy of categories. Algol and Fortran are too abstract for systems-level work; they're *problem-oriented languages*, the sort we use for solving problems in engineering or science or business. Programmers who wanted to write systems software still had to rely on their machine's assembler. After a few years of this drudgery some systems people took a step back, or, in terms of our hierarchy, a step down, and created the category of *machine-oriented languages.* As we saw when we traced C's genealogy, BCPL and B belong to this class of very-low-level software tools. Such languages are excellent for down-on-the-machine programming, but not much use for anything else—they're just too closely wedded to the computer. C is a step above them, and a step below most problem-solving languages, which is what we mean by saying that it's both high- and low-level. It fits into a very cozy niche in the hierarchy, one that somehow feels just right to many software engineers. It's close enough to the computer to give the programmer great control over the details of his program's implementation, yet far enough away that it can ignore the details of the hardware.

What C Isn't

To begin with, it isn't a language. We call it a language because everyone else does, but the analogy between human speech and programming isn't very apt. C or any other "programming language" is a set of symbols whose possible combinations are precisely defined and which can be used to represent and transform numerically coded values. If that makes C a language then musical notation is a language too, and so is algebra. We know this metaphor has great poetic appeal—math is "the language of science," music is "the universal language"—and we shall speak of "the C language" throughout this book, but understand that we're taking poetic license. Fanciful analogies have a way of hardening into laws of nature.

C isn't a branch of mathematics either, though a C program will often look like something out of an algebra text. Some new programmers stay away from C because it looks like math to them, but that's a nonproblem. You can use C to the full without knowing anything more arcane than `a = (b + 1) / c`. Because C is a relatively low-level language it knows no higher math. It stays close to the computer, which can handle only very simple arithmetic.

C isn't a religion. Some programming languages are, complete with a priesthood and a flock of disciples. So far C has escaped this kind of silliness, probably because it was designed as a tool for use by professionals who understand that no tool can be perfect.

C isn't perfect. Everyone who works with a tool swears at it sometimes, and you'll find specific criticisms of C scattered throughout this book. We can sum them up in advance by saying that C trades some elegance and some safety features for speed and ease of use. Once you're familiar with the language you'll probably prefer it that way. Since you're not familiar with it yet, we'll help you through the tricky parts.

Compiling C Programs

If this introduction has done its job, you should be convinced by now that the C language is easily accessible to human beings. Unfortunately, it's not accessible to computers, not directly: a computer can only execute the instructions built into it, instructions that programmers have to deal with at the assembly-language level. To put C into practice we need a program that translates C-language instructions into their machine-level equivalents. Such programs are called *compilers.*

In order to make use of any compiler it is first necessary to write a program in the compiler's language. When we write a program in C, we're writing what is called *source code.* The compiler's job is to take our source code and translate it into instructions that our computer can understand and execute. The compiler's output is called *executable code.* In other words, it's our program in a form that can be directly executed by our computer. Different makes of computers require different versions of the C compiler, since each has its own machine language. The source code always remains the same, but the executable code will change for each computer our program runs on.

The source code passes through a number of intermediate stages before it turns into executable code. We will assume the simplest possible case: a small program that is complete in itself. The scenario usually goes like this. A programmer logs onto his computer and, using the system editor, writes a program, which he saves as a named file. This is called the *source code* file. He then sets the compilation process in motion by typing some command—in UNIX it's `cc`. This action triggers a whole cascade of translation programs, each of which takes the user's code, translates it into a lower-level form, and passes that version along to the next translator. Here's how we might represent the cascade graphically:

$$\begin{tikzcd}
\text{Write program:} & \text{Editor} \arrow[d] \\
 & \textit{C source code} \arrow[d] \\
\text{Compile program:} & \text{C preprocessor} \arrow[d] \\
 & \begin{array}{c}\textit{expanded C}\\ \textit{source code}\end{array} \arrow[d] \\
 & \text{C compiler} \arrow[d] \\
 & \begin{array}{c}\textit{assembly-language}\\ \textit{code}\end{array} \arrow[d] \\
 & \text{Assembler} \arrow[d] \\
 & \begin{array}{c}\textit{object code}\\ \textit{from this program and}\\ \textit{library files}\end{array} \arrow[d] \\
 & \text{Linker} \arrow[d] \\
 & \textit{executable code} \arrow[d] \\
\text{Run program:} & \text{Loader}
\end{tikzcd}$$

The *C preprocessor* expands certain shorthand forms in the source code, as described in Chapter 8. Its output, the expanded source code, is fed to the *C compiler* proper. What comes out of the compiler is the original program translated into the computer's native *assembly language.* This new file is passed along to the system's assembler, which is a program that translates it into a form called *relocatable object code.* Object code is an intermediate form; it can't be read by the programmer and it can't be run by the computer. So why bother with it? Because all C programs must be linked with support routines from the *C run-time library*, which is described in Chapter 13. The *linker* performs that chore, linking all the necessary code together and translating it into an *executable code* file. The programmer can run that code by giving it to the system's loader, something that's done in UNIX simply by typing the file's name.

It's a pretty long way, then, from writing a program to running it. Luckily, we don't really have to think much about the steps involved, certainly not if we're beginners. The compilation process is hidden away, at least in UNIX. We merely type `cc` and wait a few seconds, wondering why this supposedly fast machine is so slow. At the end of those seconds we're presented with a runnable program which may or may not run the way we think it should. If it doesn't, we try to find the problem in the source code, use the editor to change it, and then compile it all over again. This happens often.

Let's run through an example. But before we do, we as authors must face up to a problem that all books on programming languages encounter. In our examples, what kind of system should we assume our readers have? The easiest way out, and the most natural, is to assume that they have exactly the same system we used to write our examples on: a PDP-11/45 running under Version 6 of UNIX. Throughout this book we shall refer to this system as our "reference computer," and to the UNIX/6 version of C as our "reference compiler." If you have access to another version of UNIX, whether Version 7 or System 3, you will find only minor differences between the examples in this book and those run on your computer. If you're using a non-UNIX version of C, there may be important differences; you should refer to your user's manual for details.

Now the example. Suppose we want to write a program that prints the words "Hell is filled with amateur musicians." We first invoke our system editor and write the source code, which looks like this:

```
main()
   {
   printf("Hell is filled with amateur musicians.\n");
   }
```