# The C Primer

Second Edition

Les Hancock

Morris Krieger

# The C Primer

Second Edition

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

Hankcock, Les.
The C primer.

Includes index.
1. C (Computer program language) I. Krieger, Morris, date. II. Title.
QA76.73.C15H36 1985 005.13'3 85-18224
ISBN 0-07-025995-X

DR87/22



123456789 DOC/DOC 89876

ISBN 0-07-025995-X

*This book was set by Coach House Press from troff copy provided by the authors. The copy was generated by a Linotron 101 driven by a Pixel 80 computer.*

*Printed and bound by R. R. Donnelley & Sons Company.*

Terms such as "man," "he," or "his" appearing in the text of this book should be understood in a generic sense. They are used solely to facilitate communication by avoiding the grammatically awkward "man/woman," "she/he," or "his/her" types of construction.

# Introduction

We've revised *The C Primer* so completely that this edition is essentially a new book. The first edition consisted of thirteen chapters; this revision contains eighteen. There were 100 complete program examples in the first edition; this revision has 129, not counting the innumerable examples of code that illustrate particular points about C. Six of the eighteen chapters (10, 14, 15, 16, 17 and 18) are entirely new. Three chapters (5, 8 and 13) have been expanded to include information omitted from the first edition, or to describe features newly added to C. Most of the other chapters have been reworked so thoroughly that hardly a paragraph remains the same.

However, our goals haven't changed, nor has our intended audience. As we said in our first edition:

> 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.

For the most part our revisions are based on reader feedback, much of which came from introductory C classes given by one of us at Bell Labs and Bellcore using *The C Primer* as the textbook. One effect of our classroom experience was that we threw out our original chapter on C pointers. Though we're far from claiming to say the last word on

pointers, we believe our revised presentation in Chapters 10 and 16 provides a much more useful, pragmatic introduction.

This edition also includes an extended treatment of file I/O, a topic relegated to the last few pages of the original edition. As we discuss in Chapters 11 and 17, the I/O functions are not, strictly speaking, a part of the C language. They belong to a library of functions supplied with the C compiler. When we began work on the book in 1980, we were reluctant to deal with these library functions; they seemed more a part of the UNIX* operating system than of the C language. Over the last few years, however, dozens of C compilers have been written for use in non-UNIX environments, and nearly every vendor has taken pains to supply a library of functions that follow the UNIX standard. In fact, these functions are included in the C standard proposed by the American National Standards Institute. They've become a de facto part of the language, and we feel we can now treat them as such.

Standards loom large in the mind of anyone who tries to write a book on any computer language. When a language has been in use for more than a year or two, it's certain to exist in several versions, depending on the machines it runs on. Which compiler and which machine should we assume our readers have? What we've done is base our examples on the C compiler delivered with UNIX System V for the DEC PDP-11/70. Throughout the book we refer to these as our "reference compiler" and "reference computer." At the same time, we've tried to make our examples as plain as possible, to present C code that will run on almost any installation. We have, in fact, tested our examples on a variety of machines. The differences in output have been insignificant. It is only when we discuss bit manipulation in Chapters 14 and 15 that differences in machine output become important, and we point these differences out as we come to them. Otherwise, the reader's understanding of the text and the examples shouldn't be affected at all by whatever C compiler and computer he is using.

Again, we'd like to acknowledge the help of our co-workers and students at Bell Labs and Bellcore. We would also like to thank Dr. Melvin Ferentz, Director of Computing Services at The Rockefeller University, for permission to use the computing facilities of that university in preparing this text, and, for all their help, Dr. Banvir Chaudhary and Armand Gazes, both of The Rockefeller University, Tony Doblmaier, Distinguished MTS and recent retiree from Bellcore, and Peter Dunn of P. S. Dunn Associates.

* UNIX is a trademark of AT&T Bell Laboratories.

# Table of Contents

Chapter

# 1

# What C Is

C is a programming language developed at AT&T 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 C compiler and almost all of the operating system, 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. Today there are dozens of C compilers available, many of them running on 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 made 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 simplification of CPL—and a very

* 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.

spare language it is indeed, though suited for use on the hardware available to Thompson. 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 main 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—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, asked 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 machine-oriented to the 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 assembly languages force you to think in terms of the hardware and 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 first 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, yet still 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 to 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 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.

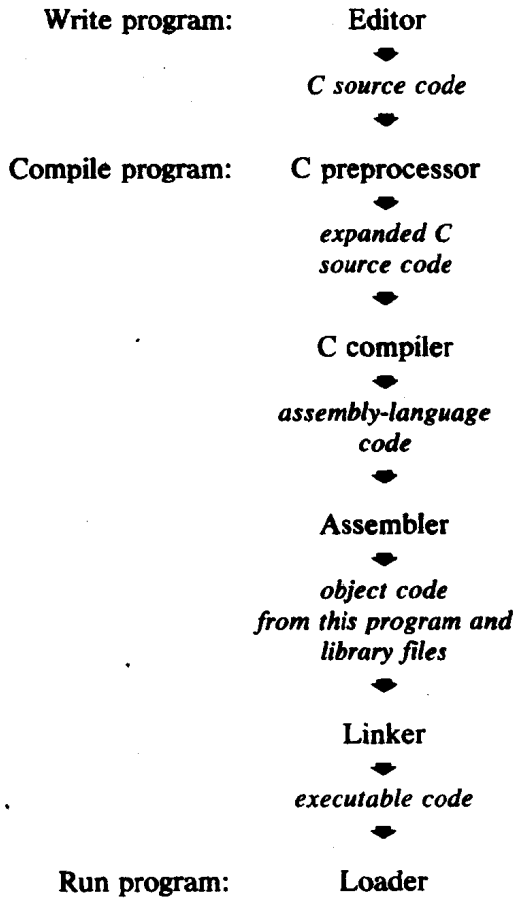
## Compiling C Programs

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 the C language, what we write 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 make will have its own machine language. The source code remains always 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 pos-

sible 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 called the *source code file*. He then sets the compilation process in motion by typing the appropriate command—in UNIX it's cc. This action triggers a whole cascade of translation programs, each of which takes the programmer's source 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:




The *C preprocessor* expands certain shorthand forms in the source code, as we will describe 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 *assembly language*. The assembly language version is passed along to the

system's assembler, which 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*. The *linker* performs this chore, linking all the necessary code together and converting 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 when he types 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 plus the filename 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 make a change, 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: a 16-bit computer running under UNIX V. Throughout this book we shall refer to this system as our "reference computer," and to the UNIX/ V version of C as our "reference compiler." If you have access to another version of UNIX, 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 for 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 following source code:

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

Let's suppose we save this source code under the filename test1_1.c. All we need do to compile and execute the program on our reference computer is enter the commands shown on the first two lines below:*

---

* In these and all the other programming examples in this book, all commands are executed only when you press the carriage-return key after having entered the command. What the user types will always be shown in **boldface**.

```
$ cc test1_1.c
$ a.out
Hell is filled with amateur musicians.
```

The first thing we should mention is that the *prompt character* on our reference computer is $. This prompt must appear at the terminal before we can enter any command. So, following the $, we enter the compile command cc followed by the name of the source code file:

```
$ cc test1_1.c
```

Our filename must have the suffix .c. This is true of all C source code filenames. If the suffix is missing, the program won't be compiled.

Assuming the compilation proceeds smoothly to its conclusion, the system prompt $ will again appear on the terminal. We can now execute the compiled code by typing a.out after the prompt:

```
$ a.out
```

and the computer will immediately run the program, displaying the sentence at our terminal:

```
Hell is filled with amateur musicians.
```

If you have been following this procedure on your own machine, and if you now examine your file directory, you will see that your directory contains a new file, a.out. You can execute the file a.out any time you wish by typing the command a.out after the system prompt.

The UNIX C compiler always puts its output into the file a.out. Of course you can change the file's name to anything you like and run the program by typing its new name instead.

This is the basic compilation procedure. It should work on every standard UNIX installation just as we've described it. Now for a few variations. Our program is very short, and all the source code can be kept in a single file. But real-life C programs are much larger than this. They usually consist of several source files, each of which has been separately compiled and stored in memory under its own filename until the programmer is ready to collect them into a complete program. As an example of how such a set of files is compiled and saved, we'll use a variant of our previous program. This variant looks similar, but it's not a complete C program—it doesn't begin with main. (Don't worry about the form of these example programs; all will be explained in due course):