CHAPTER 0: **INTRODUCTION**

C is a general-purpose programming language. It has been closely asso­   
ciated with the UNIX system, since it was developed on that system, and   
since UNIX and its software are written in C. The language, however, is not   
tied to any one operating system or machine; and although it has been called   
a "system programming language" because it is useful for writing operating   
systems, it has been used equally well to write major numerical, text-   
processing, and data-base programs.

C is a relatively "low level" language. This characterization is not   
pejorative; it simply means that C deals with the same sort of objects that   
most computers do, namely characters, numbers, and addresses. These may   
be combined and moved about with the usual arithmetic and logical opera­   
tors implemented by actual machines.

C provides no operations to deal directly with composite objects such as   
character strings, sets, lists, or arrays considered as a whole. There is no   
analog, for example, of the **PIM** operations which manipulate an entire   
array or string. The language does not define any storage allocation facility   
other than static definition and the stack discipline provided by the local   
variables of functions: there is no heap or garbage collection like that pro­   
vided by Algol 68. Finally, C itself provides no input-output facilities: there   
are no READ or WRITE statements, and no wired-in file access methods.   
All of these higher-level mechanisms must be provided by explicitly-called   
functions.

Similarly, C offers only straightforward, single-thread control flow con­   
structions: tests, loops, grouping, and subprograms, but not multiprogram­   
ming, parallel operations, synchronization, or coroutines.

Although the absence of some of these features may seem like a grave   
deficiency ("You mean I have to call a function to compare two character   
strings?"), keeping the language down to modest dimensions has brought   
real benefits. Since C is relatively small, it can be described in a small   
space, and learned quickly. A compiler for C can be simple and compact.   
Compilers are also easily written; using current technology, one can expect   
to prepare a compiler for a new machine in a couple of months, and to find

1

2 ' THE C PROGRAMMING LANGUAGE CHAPTER 0

that 80 percent of the code of a new compiler is common with existing ones.   
This provides a high degree of language mobility. Because the data types   
and control structures provided by C are supported directly by most existing   
computers, the run-time library required to implement self-contained pro­   
grams is tiny. On the PDP-11, for example, it contains only the routines to   
do 32-bit multiplication and division and to perform the subroutine entry   
and exit sequences. Of course, each implementation provides a comprehen­   
sive, compatible library of functions to carry out I/O, string handling, and   
storage allocation operations, but since they are called only explicitly, they   
can be avoided if required; they can also be written portably in C itself.

Again because the language reflects the capabilities of current comput­   
ers, C programs tend to be efficient enough that there is no compulsion to   
write assembly language instead. The most obvious example of this is the   
UNIX operating system itself, which is written almost entirely in C. Of   
13000 lines of system code, only about 800 lines at the very lowest level are   
in assembler. In addition, essentially all of UNIX applications software is   
written in C; the vast majority of UNIX users (including one of the authors   
of this book) do not even know the PDP-11 assembly language.

Although C matches the capabilities of many computers, it is indepen­   
dent of any particular machine architecture, and so with a little care it is   
easy to write "portable" programs, that is, programs which can be run   
without change on a variety of hardware. It is now routine in our environ­   
ment that software developed on UNIX is transported to the local   
Honeywell, IBM and Interdata systems. In fact, the C compilers and run­   
time support on these four machines are much more compatible than the   
supposedly ANSI standard versions of Fortran. The UNIX operating system   
itself now runs on both the PDP-11 and the Interdata 8/32. Outside of pro­   
grams which are necessarily somewhat machine-dependent like the compiler,   
assembler, and debugger, the software written in C is identical on both   
machines. Within the operating system itself, the 7000 lines of code outside   
of the assembly language support and the I/O device handlers is about 95   
percent identical.

For programmers familiar with other languages, it may prove helpful to   
mention a few historical, technical, and philosophical aspects of C, for con­   
trast and comparison.

Many of the most important ideas of C stem from the considerably   
older, but still quite vital, language BCPL, developed by Martin Richards.   
The influence of BCPL on C proceeded indirectly through the language B,   
which was written by Ken Thompson in 1970 for the first UNIX system on   
the PDP-7.

Although it shares several characteristic features with **BCPL, C** is in no   
sense a dialect of it. BCPL and B are "typeless" languages: the only data   
type is the machine word, and access to other kinds of objects is by special

CHAPTER 0 INTRODUCTION 3

operators or function calls. In C, the fundamental data objects are charac­   
ters, integers of several sizes, and floating point numbers. In addition, there   
is a hierarchy of derived data types created with pointers, arrays, structures,   
unions, and functions.

C provides the fundamental flow-control constructions required for   
well-structured programs: statement grouping; decision making (if); loop­   
ing with the termination test at the top **(while, for),** or at the bottom   
(do); and selecting one of a set of possible cases (switch). (All of these   
were provided in **BCPL** as well, though with somewhat different syntax; that   
language anticipated the vogue for "structured programming" by several   
years.)

C provides pointers and the ability to do address arithmetic. The argu­   
ments to functions are passed by copying the value of the argument, and it   
is impossible for the called function to change the actual argument in the   
caller. When it is desired to achieve "call by reference," a pointer may be   
passed explicitly, and the function may change the object to which the   
pointer points. Array names are passed as the location of the array origin,   
so array arguments are effectively call by reference.

Any function may be called recursively, and its local variables are typi­   
cally "automatic," or created anew with each invocation. Function   
definitions may not be nested but variables may be declared in a block-   
structured fashion. The functions of a C program may be compiled   
separately. Variables may be internal to a function, external but known only   
within a single source file, or completely global. Internal variables may be   
automatic or static. Automatic variables may be placed in registers for   
increased efficiency, but the register declaration is only a hint to the com­   
piler, and does not refer to specific machine registers.

C is not a strongly-typed language in the sense of Pascal or Algol 68. It   
is relatively permissive about data conversion, although it will not automati­   
cally convert data types with the wild abandon of **PL/I.** Existing compilers   
provide no run-time checking of array subscripts, argument types, etc.

For those situations where strong type checking is desirable, a separate   
version of the compiler is used. This program is called *lint,* apparently   
because it picks bits of fluff from one's programs. *lint* does not generate   
code, but instead applies a very strict check to as many aspects of a program   
as can be verified at compile and load time. It detects type mismatches,   
inconsistent argument usage, unused or apparently uninitialized variables,   
potential portability difficulties, and the like. Programs which pass   
unscathed through *lint* enjoy, with few exceptions, freedom from type errors   
about as complete as do, for example, Algol 68 programs. We will mention   
other *lint* capabilities as the occasion arises.

Finally, C, like any other language, has its blemishes. Some of the   
operators have the wrong precedence; some parts of the syntax could be   
better; there are several versions of the language extant, differing in minor

4 THE C PROGRAMMING LANGUAGE CHAPTER 0

ways. Nonetheless, C has proven to be an extremely effective and expres­   
sive language for a wide variety of programming applications.

The rest of the book is organized as follows. Chapter 1 is a tutorial   
introduction to the central part of C. The purpose is to get the reader   
started as quickly as possible, since we believe strongly that the only way to   
learn a new language is to write programs in it. The tutorial does assume a   
working knowledge of the basic elements of programming; there is no expla­   
nation of computers, of compilation, nor of the meaning of an expression   
like n=n+1. Although we have tried where possible to show useful pro­   
gramming techniques, the book is not intended to be a reference work on   
data structures and algorithms; when forced to a choice, we have concen­   
trated on the language.

Chapters 2 through 6 discuss various aspects of C in more detail, and   
rather more formally, than does Chapter 1, although the emphasis is still on   
examples of complete, useful programs, rather than isolated fragments.   
Chapter 2 deals with the basic data types, operators and expressions.   
Chapter 3 treats control flow: if—else, while, for, etc. Chapter 4 cov­   
ers functions and program structure — external variables, scope rules, and   
so on. Chapter 5 discusses pointers and address arithmetic. Chapter 6 con­   
tains the details of structures and unions.

Chapter 7 describes the standard C I/O library, which provides a com­   
mon interface to the operating system. This I/O library is supported on all   
machines that support C, so programs which use it for input, output, and   
other system functions can be moved from one system to another essentially   
without change.

Chapter 8 describes the interface between C programs and the UNIX   
operating system, concentrating on input/output, the file system, and porta­   
bility. Although some of this chapter is UNIX-specific, programmers who   
are not using a UNIX system should still find useful material here, including   
some insight into how one version of the standard library is implemented,   
and suggestions on achieving portable code.

Appendix A contains the C reference manual. This is the "official"   
statement of the syntax and semantics of C, and (except for one's own com­   
piler) the final arbiter of any ambiguities and omissions from the earlier   
chapters.

Since C is an evolving language that exists on a variety of systems, some   
of the material in this book may not correspond to the current state of   
development for a particular system. We have tried to steer clear of such   
problems, and to warn of potential difficulties. When in doubt, however, we   
have generally chosen to describe the PDP-11 UNIX situation, since that is   
the environment of the majority of C programmers. Appendix A also   
describes implementation differences on the major C systems.