

CHAPTER 0: INTRODUCTION

C is a general-purpose programming language. It has been closely associated 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 operators 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 PL/I 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 provided 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.

The lack of a "heap" or "garbage collection" feature in C is both one of the great strengths of the language and at the same time is likely reason that the average programmer will never develop or maintain a major C application during their career.

C provides a simple feature using the `malloc()` and `free()` functions that allows a programmer to request a certain amount of memory be allocated dynamically, use the memory and then return the memory to the C runtime library for later reuse. For example to convert a JPG image to a PNG image, our application will read the JPG data into memory, then convert the image into a PNG image in memory and then write the PNG data out to a file. We don't know how large the images will be in advance, so we request whatever size we need from C and then give it back when we are done.

The term "heap" refers to memory that C manages on our behalf when we need to "borrow" a bit of memory and give it back later. There are a couple of issues with a simple heap implementation. First, if we "forget" to call `free()` when we are done with the memory, we have created a "memory leak" and our program eventually will run out of memory and abort. C places the onus of giving back any dynamically allocated memory on the programmer. Modern languages like Java, JavaScript, and Python keep track of when we stop using a segment of dynamic memory using a layer that can automatically reclaim the memory.

The more difficult problem is after a series of calls to `malloc()` and `free()` the heap space becomes fragmented and some cleanup is needed. This clean up is called "garbage collection". Efficient memory allocation and garbage collection has been the subject of decades of computer science research. The Java language has built a number of increasingly effective garbage collection approaches over the years.

Kernighan and Ritchie in one simple paragraph define most of the problem as out of scope for the C language. Which makes it a bit challenging for us to make good use of dynamic memory allocation in C - but when we do it properly - it performs very well.

If you are using a language like Java, Python, or PHP, every time you create a new string through concatenation without thinking about memory allocation, remember to appreciate the decades of work by computer scientists that made it easy for you. Kernighan and Ritchie knew "garbage collection" was difficult. So they left it out of the C language and put it into a run-time library.

Similarly, C offers only straightforward, single-thread control flow constructions: tests, loops, grouping, and subprograms, but not multiprogramming, 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 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 programs 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 comprehensive, 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 computers, 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.

In this preface, the authors are carefully explaining the fact that many of the well-established programming languages of the 1960's and 1970's like FORTRAN, COBOL,

Pascal, Algol, and PL/I were solving many of the use cases that were needed by programmers by adding syntax to the languages. The creators of C and UNIX were advocating for a more minimal set of programming language constructs and more reliance on calling functions in provided run-time libraries to meet programmer use cases. It may have seemed a strange approach to experienced programmers in the 1980's, but over time it has allowed C to expand to meet a wide range of programmer needs without requiring major revisions to the core language or compiler.

Although C matches the capabilities of many computers, it is independent 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 environment that software developed on UNIX is transported to the local Honeywell, IBM and Interdata systems. In fact, the C compilers and runtime 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 programs 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.

Before UNIX and C, if you were running the vendor operating system and writing in the best language for systems like the PDP/11 and Interdata 8/32 - the user experience was *completely* different. Today we take for granted that expect to be able to download the same application for a Windows, MacOS, or a Linux system. Even in the 1970's those that were using UNIX and C could write code once and move it between two hardware platforms and expect that it would work with no or relatively few changes.

For programmers familiar with other languages, it may prove helpful to mention a few historical, technical, and philosophical aspects of C, for contrast 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 operators or function calls. In C, the fundamental data objects are characters, 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`); looping 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 arguments 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 typically "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 compiler, 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 automatically 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.

Separating the "checking for things that *might* be wrong" into the `lint` program keeps the C compiler simple and easy to port to a new computer. The `lint` program was naturally a portable text-processing application. While there is some initial overlap between a `lint` program and a compiler, over time, there is quite distinct research and expertise in "how to lint" versus "how to compile". Modern `lint` programs look at programs in far more detail than most compilers. Separating the concerns of `lint` and the C compiler, also allowed `lint` programs to use more memory, and take more time to execute than compilers. Since the typical developer might use the compiler many times per day and run `lint` less often it was nice for the compiler to run quickly and make light use of computer resources.

We call this idea of building two smaller programs that each specialize in one task, "separation of concerns" and it is an important principle in Computer Science. By keeping each component simple and focused, we can more easily build, test, and verify each component. UNIX and C showed the benefits of taking a "many small components"

approach to solve an overall set of problems.

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 ways. Nonetheless, C has proven to be an extremely effective and expressive 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 explanation of computers, of compilation, nor of the meaning of an expression like $n=n+1$. Although we have tried where possible to show useful programming techniques, the book is not intended to be a reference work on data structures and algorithms; when forced to a choice, we have concentrated 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](#) covers functions and program structure - external variables, scope rules, and so on. [Chapter 5](#) discusses pointers and address arithmetic. [Chapter 6](#) contains the details of structures and unions.

[Chapter 7](#) describes the standard C I/O library, which provides a common 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.

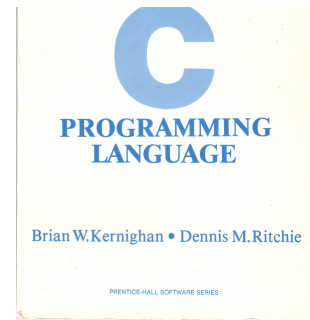
[Chapter 8](#) describes the interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and portability. Although some of this chapter is UNIX-specific, programmers who are not using a UNIX system should still find useful material here, including some 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 compiler) 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.

Their book is copyright all rights reserved by AT&T but is being used in this work under "fair use" because of the book's historical and scholarly significance, its lack of availability, and lack of an accessible version of the book.

The book is augmented in places to help understand its rightful place in a historical context amidst the major changes of the 1970's and 1980's as Computer Science evolved from a hardware-first / vendor-centered approach to a software-centered approach where portable operating systems and applications written in C could run on any hardware.



This is not the ideal book to learn C programming because the 1978 edition does not reflect the modern C language. Using an obsolete book gives us an opportunity to take students back in time and understand how the C language was evolving as it laid the ground work for a future with portable applications.

If you want to learn modern C programming from a book that reflects the modern C language, I suggest you use the [C Programming Language, 2nd Edition](#), also written by Brian W. Kernighan and Dennis M. Ritchie, initially published in 1988 and based on the [ANSI C](#) (C89) version of the language.

The source code to this book is at <https://github.com/csev/cc4e>. You are welcome to help in the creation and editing of the book.