The C Book

Mike Banahan, Declan Brady, Mark Doran

The C Book

Mike Banahan, Declan Brady, Mark Doran

Contents

Preface

About This Book

This book was written with two groups of readers in mind. Whether you are new to C and want to learn it, or already know the older version of the language but want to find out more about the new standard, we hope that you will find what follows both instructive and at times entertaining too. This is not a tutorial introduction to programming. The book is designed for programmers who already have some experience of using a modern highlevel procedural programming language. As we explain later, C isn't really appropriate for complete beginners–though many have managed to use it– so the book will assume that its readers have already done battle with the notions of statements, variables, conditional execution, arrays, procedures (or subroutines) and so on. Instead of wasting your time by ploughing through tedious descriptions of how to add two numbers together and explaining that the symbol for multiplication is *****, the book concentrates on the things that are special to C. In particular, it's the **way** that C is used which is emphasized. Those who already know C will be interested in the new Standard and how it affects existing C programs. The effect on existing programs might not at first seem to be important to newcomers, but in fact the 'old' and new versions of the language **are** an issue for the beginner too. For some years after the approval of the Standard, programmers will have to live in a world where they can easily encounter a mixture of both the new and the old language, depending on the age of the programs that they are working with. For that reason, the book highlights where the old and new features differ significantly. Some of the old features are no ornament to the language and are well worth avoiding; the Standard goes so far as to consider them obsolescent and recommends that they should not be used. For that reason they are not described in detail, but only far enough to allow a reader to understand what they mean. Anybody who intends to **write** programs using these old-style features should be reading a different book.

This is the second edition of the book, which has been revised to refer to the final, approved version of the Standard. The first edition of the book was based on a draft of the Standard which did contain some differences from the draft that was eventually approved. During the revision we have taken the opportunity to include more summary material and an extra chapter illustrating the use of C and the Standard Library to solve a number of small problems.

The Success of C

C is a remarkable language. Designed originally by one man, Dennis Ritchie, working at AT&T Bell Laboratories in New Jersey, it has increased in use until now it may well be one of the most widely-written computer languages in the world. The success of C is due to a number of factors, none of them key, but all of them important. Perhaps the most significant of all is that C was developed by real practioners of programming and was designed for practical day-to-day use, not for show

or for demonstration. Like any well-designed tool, it falls easily to the hand and feels good to use. Instead of providing constraints, checks and rigorous boundaries, it concentrates on providing you with power and on not getting in your way.

Because of this, it's better for professionals than beginners. In the early stages of learning to program you need a protective environment that gives feedback on mistakes and helps you to get results quickly—programs that run, even if they don't do what you meant. C is not like that! A professional forester would use a chain-saw to cut down trees quickly, aware of the dangers of touching the blade when the machine is running; C programmers work in a similar way. Although modern C compilers do provide a limited amount of feedback when they notice something that is out of the ordinary, you almost always have the option of forcing the compiler to do what you said you wanted and to stop it from complaining. Provided that what you said you wanted was what you really did want, then you'll get the result you expected. Programming in C is like eating red meat and drinking strong rum except your arteries and liver are more likely to survive it.

Not only is C popular and a powerful asset in the armoury of the serious day-to-day programmer, there are other reasons for the success of this language. It has always been associated with the UNIX operating system and has benefited from the increasing popularity of that system. Although it is not the obvious first choice for writing large commercial data processing applications, C has the great advantage of always being available on commercial UNIX implementations. UNIX is written in C, so whenever UNIX is implemented on a new type of hardware, getting a C compiler to work for that system is the first task. As a result it is almost impossible to find a UNIX system without support for C, so the software vendors who want to target the UNIX marketplace find that C is the best bet if they want to get wide coverage of the systems available. Realistically, C is the first choice for portability of software in the UNIX environment.

C has also gained substantially in use and availability from the explosive expansion of the Personal Computer market. C could almost have been designed specifically for the development of software for the PC—developers get not only the readability and productivity of a high-level language, but also the power to get the most out of the PC architecture **without** having to resort to the use of assembly code. C is practically unique in its ability to span two levels of programming; as well as providing high-level control of flow, data structures and procedures—all of the stuff expected in a modern high-level language—it also allows systems programmers to address machine words, manipulate bits and get close to the underlying hardware if they want to. That combination of features is very desirable in the competitive PC software markeplace and an increasing number of software developers have made C their primary language as a result.

Finally, the extensibility of C has contributed in no small way to its popularity. Many other languages have failed to provide the file access and general input-output features that are needed for industrial-strength applications. Traditionally, in these languages I/O is built-in and is actually understood by the compiler. A master-stroke in the design of C (and interestingly, one of the strengths of the UNIX system too) has been to take the view that if you don't know how to provide a complete solution to a generic requirement, instead of providing half a solution (which invariably pleases nobody), you should allow the users to build their own. Software designers the world over have something to learn from this! It's the approach that has been taken by C, and not only for I/O. Through the use of

library functions you can extend the language in many ways to provide features that the designers didn't think of. There's proof of this in the so-called Standard I/O Library (stdio), which matured more slowly than the language, but had become a sort of standard all of its own before the Standard Committee give it official blessing. It proved that it is possible to develop a model of file I/O and associated features that is portable to many more systems than UNIX, which is where it was first wrought. Despite the ability of C to provide access to low-level hardware features, judicious style and the use of the stdio package results in highly portable programs; many of which are to be found running on top of operating systems that look very different from one another. The nice thing about this library is that if you don't like what it does, but you have the appropriate technical skills, you can usually extend it to do what you do want, or bypass it altogether.

Standards

Remarkably, C achieved its success in the absence of a formal standard. Even more remarkable is that during this period of increasingly widespread use, there has never been any serious divergence of C into the number of dialects that has been the bane of, for example, BASIC. In fact, this is not so surprising. There has always been a "language reference manual", the widely-known book written by Brian Kernighan and Dennis Ritchie, usually referred to as simply "K&R".

The C Programming Language,

B.W. Kernighan and D. M. Ritchie,

Prentice-Hall

Englewood Cliffs,

New Jersey,

1978

Further acting as a rigorous check on the expansion into numerous dialects, on UNIX systems there was only ever really one compiler for C; the so-called "Portable C Compiler", originally written by Steve Johnson. This acted as a reference implementation for C—if the K&R reference was a bit obscure then the behaviour of the UNIX compiler was taken as the definition of the language. Despite this almost ideal situation (a reference manual and a reference implementation are extremely good ways of achieving stability at a very low cost), the increasing number of alternative implementations of C to be found in the PC world did begin to threaten the stability of the language.

The X3J11 committee of the American National Standards Institute started work in the early 1980's to produce a formal standard for C. The committee took as its reference the K&R definition and began its lengthy and painstaking work. The job was to try to eliminate ambiguities, to define the undefined, to fix the most annoying deficiencies of the language and to preserve the spirit of C—all this as well as providing as much compatibility with existing practice as was possible. Fortunately, nearly all of the developers of the competing versions of C were represented on the committee, which in itself acted as a strong force for convergence right from the beginning.

Development of the Standard took a long time, as standards often do. Much of the work is not just technical, although that is a very time-consuming part of the job, but also procedural. It's easy to underrate the procedural aspects of standards work, as if it somehow dilutes the purity of the technical work, but in fact it is equally important. A standard that has no agreement or consensus in the industry is unlikely to be widely adopted and could be useless or even damaging. The painstaking work of obtaining consensus among committee members is critical to the success of a practical standard, even if at times it means compromising on technical "perfection", whatever that might be. It is a democratic process, open to all, which occasionally results in aberrations just as much as can excessive indulgence by technical purists, and unfortunately the delivery date of the Standard was affected at the last moment by procedural, rather than technical issues. The technical work was completed by December 1988, but it took a further year to resolve procedural objections. Finally, approval to release the document as a formal American National Standard was given on December 7th, 1989.

Hosted and Free-Standing Environments

he dependency on the use of libraries to extend the language has an important effect on the practical use of C. Not only are the Standard I/O Library functions important to applications programmers, but there are a number of other functions that are widely taken almost for granted as being part of the language. String handling, sorting and comparison, character manipulation and similar services are invariably expected in all but the most specialized of applications areas.

Because of this unusually heavy dependency on libraries to do real work, it was most important that the Standard provided comprehensive definitions for the supporting functions too. The situation with the library functions was much more complicated than the relatively simple job of providing a tight definition for the language itself, because the library can be extended or modified by a knowledgeable user and was only partially defined in K&R. In practice, this led to numerous similar but different implementations of supporting libraries in common use. By far the hardest part of the work of the Committee was to reach a good definition of the library support that should be provided. In terms of benefit to the final user of C, it is this work that will prove to be by far and away the most valuable part of the Standard.

However, not all C programs are used for the same type of applications. The Standard Library is useful for 'data processing' types of applications, where file I/O and numeric and string oriented data are widely used. There is an equally important application area for C—the 'embedded system' area—which includes such things as process control, real-time and similar applications.

The Standard knows this and provides for it. A large part of the Standard is the definition of the library functions that must be supplied for hosted environments. A hosted environment is one that provides the standard libraries. The standard permits both hosted and freestanding environments. and goes to some length to differentiate between them. Who would want to go without libraries? Well, anybody writing 'stand alone' programs. Operating systems, embedded systems like machine controllers and firmware for instrumentation are all examples of the case where a hosted environment might be inappropriate. Programs written for a hosted environment

have to be aware of the fact that the names of all the library functions are reserved for use by the implementation. There is no such restriction on the programmer working in a freestanding environment, although it isn't a good idea to go using names that are used in the standard library, simply because it will mislead readers of the program. Chapter 9[1] describes the names and uses of the library functions.

Typographical conventions

The book tries to keep a consistent style in its use of special or technical terms. Words with a special meaning to C, such as reserved words or the names of *library functions*, are printed in a different typeface. Examples are `int` and `printf`. Terms used by the book that have a meaning not to C but in the Standard or the text of the book, are bold if they have not been introduced recently. They are **not** bold everywhere, because that rapidly annoys the reader. As you have noticed, italics are also used for emphasis from time to time, and to introduce loosely defined terms. Whether or not the name of a function, keyword or so on starts with a capital letter, it is nonetheless capitalized when it appears at the start of a sentence; this is one problem where either solution (capitalize or not) is unsatisfactory. Occasionally quote marks are used around 'special terms' if there is a danger of them being understood in their normal English meaning because of surrounding context. Anything else is at the whim of the authors, or simply by accident.

Order of topics

The order of presentation of topics in this book loosely follows the order that is taught in The Instruction Set's introductory course. It starts with an overview of the essential parts of the language that will let you start to write useful programs quite quickly. The introduction is followed by a detailed coverage of the material that was ignored before, then it goes on to discuss the standard libraries in depth. This means that in principle, if you felt so inclined, you could read the book as far as you like and stop, yet still have learnt a reasonably coherent subset of the language. Previous experience of C will render Chapter 1[2] a bit slow, but it is still worth persevering with it, if only once.

Example programs

All but the smallest of the examples shown in the text have been tested using a compiler that claims to conform to the Standard. As a result, most of them stand a good chance of being correct, unless our interpretation of the Standard was wrong and the compiler developer made the same mistake. None the less, experience warns that despite careful checking, **some** errors are bound to creep in. Please be understanding with any errors that you may find.

[1] http://publications.gbdirect.co.uk/c_book/chapter9/
[2] http://publications.gbdirect.co.uk/c_book/chapter1/

Deference to Higher Authority

This book is an attempt to produce a readable and enlightening description of the language defined by the Standard. It sets out to to make interpretations of what the Standard actually means but to express them in 'simpler' English. We've done our best to get it right, but you must never forget that the only place that the language is fully defined is in the Standard itself. It is entirely possible that what we interpret the Standard to mean is at times not what the Standard Committee sought to specify, or that the way we explain it is looser and less precise than it is in the Standard. If you are in any doubt: READ THE STANDARD! It's not meant to be read for pleasure, but it is meant to be accurate and unambiguous; look nowhere else for the authoritative last word.

Address for the Standard

Copies of the Standard can be obtained from: X3 Secretariat,

> CBEMA, 311 First Street, NW, Suite 500, Washington DC 20001-2178, USA.
>
> Phone (+1) (202) 737 8888

Mike Banahan

Declan Brady

Mark Doran

January 1991

An Introduction to C

The form of a C program

If you're used to the block-structured form of, say, Pascal, then at the outer level the layout of a C program may surprise you. If your experience lies in the FORTRAN camp you will find it closer to what you already know, but the inner level will look quite different. C has borrowed shamelessly from both kinds of language, and from a lot of other places too. The input from so many varied sources has spawned a language a bit like a cross-bred terrier: inelegant in places, but a tenacious brute that the family is fond of. Biologists refer to this phenomenon as 'hybrid vigour'. They might also draw your attention to the 'chimera', an artificial crossbreed of creatures such as a sheep and a goat. If it gives wool and milk, fine, but it might equally well just bleat and stink!

At the coarsest level, an obvious feature is the multi-file structure of a program. The language permits *separate compilation*, where the parts of a complete program can be kept in one or more *source files* and compiled independently of each other. The idea is that the compilation process will produce files which can then be *linked* together using whatever link editor or loader that your system provides. The block structure of the Algol-like languages makes this harder by insisting that the whole program comes in one chunk, although there are usually ways of getting around it.

The reason for C's approach is historical and rather interesting. It is supposed to speed things up: the idea is that compiling a program into relocatable *object code* is slow and expensive in terms of resources; compiling is hard work. Using the loader to bind together a number of object code modules should simply be a matter of sorting out the absolute addresses of each item in the modules when combined into a complete program. This should be relatively inexpensive. The expansion of the idea to arrange for the loader to scan *libraries* of object modules, and select the ones that are needed, is an obvious one. The benefit is that if you change one small part of a program then the expense of recompiling all of it may be avoided; only the module that was affected has to be recompiled.

All, the same, it's true that the more work put on to the loader, the slower it becomes, in fact sometimes it can be the slowest and most resource consuming part of the whole procedure. It is possible that, for some systems, it would be quicker to recompile everything in one go than to have to use the loader: Ada has sometimes been quoted as an example of this effect occurring. For C, the work that has to be done by the loader is not large and the approach is a sensible one. Figure 1.1 shows the way that this works.

<image 1.1>

Figure 1.1. Separate compilation

This technique is important in C, where it is common to find all but the smallest of programs constructed from a number of separate source files. Furthermore, the extensive use that C makes of libraries means that even trivial programs pass through the loader, although that might not be obvious at the first glance or to the newcomer.

Functions

A C program is built up from a collection of items such as *functions* and what we could loosely call *global variables*. All of these things are given names at the point where they are defined in the program; the way that the names are used to access those items from a given place in the program is governed by rules. The rules are described in the Standard using the term *linkage*. For the moment we only need to concern ourselves with *external linkage* and *no linkage*. Items with external linkage are those that are accessible throughout the program (library functions are a good example); items with no linkage are also widely used but their accessibility is much more restricted. Variables used inside functions are usually 'local' to the function; they have no linkage. Although this book avoids the use of complicated terms like those where it can, sometimes there isn't a plainer way of saying things. Linkage is a term that you are going to become familiar with later. The only external linkage that we will see for a while will be when we are using functions.

Functions are C's equivalents of the functions and subroutines in FORTRAN, functions and procedures in Pascal and ALGOL. Neither BASIC in most of its simple mutations, nor COBOL has much like C's functions.

The idea of a function is, of course, to allow you to encapsulate one idea or operation, give it a name, then to call that operation from various parts of the rest of your program simply by using the name. The detail of what is going on is not immediately visible at the point of use, nor should it be. In well designed, properly structured programs, it should be possible to change the way that a function does its job (as long as the job itself doesn't change) with no effect on the rest of the program.

In a *hosted environment* there is one function whose name is special; it's the one called `main`. This function is the first one entered when your program starts running. In a *freestanding environment* the way that a program starts up is *implementation defined*; a term which means that although the Standard doesn't specify what must happen, the actual behaviour must be consistent and documented. When the program leaves the main function, the whole program comes to an end. Here's a simple program containing two functions:

```
1  #include <stdio.h>
2
3  /*
4  * Tell the compiler that we intend
5  * to use a function called show_message.
6  * It has no arguments and returns no value
7  * This is the "declaration".
8  *
9  */
10
11 void show_message(void);
12 /*
13 * Another function, but this includes the body of
```

```
14  * the function. This is a "definition".
15  */
16  main(){
17       int count;
18
19       count = 0;
20       while(count < 10){
21                show_message();
22                count = count + 1;
23       }
24
25       return(0);
26  }
27
28  /*
29  * The body of the simple function.
30  * This is now a "definition".
31  */
32  void show_message(void){
33       printf("hello\n");
34  }
```

Example 1.1