The C Programming Language, or K&R as it is affectionately known, is widely praised by experienced C programmers as one of the best books on C there is. (It was also the first; it also happens to be a best-seller.) The only real criticism K&R ever receives is that it may not be the best tutorial for beginners; it seems to assume a certain amount of programming savvy and familiarity with computers. Actually, if you read it carefully, you’ll find that it is constantly dispensing wisdom about programming in general, from basic concepts to deep insights to impeccable commentary on imponderable topics such as programming style, at the same time it teaches the specifics of the C language. Therefore, the fundamental criticism may simply be that K&R is not suitable for those who read carelessly.

The authors are not out to save the world or to convert it to their philosophy of programming. When they say something, they say it once, without theatrics or undue emphasis. If you read the book too quickly, or skim it, or look only for specific answers to what you think you’re trying to learn today, you will miss much of the excellent advice which the authors have to offer.

These notes were prepared (beginning in Spring, 1995) for the University of Washington Experimental College course in Introductory C Programming. They are meant to supplement K&R for the reader who is new to C and perhaps to programming, and who wants a slightly more detailed, less pithy presentation. I’ll add insights from my own experience, in particular by pointing out those areas where people traditionally misunderstand something about C or K&R’s presentation of it. I’ll also call out a few of the very deep sentences, which you might overlook at first even if you’re not skimming (perhaps because their significance only becomes apparent once you’ve begun writing bigger or more complicated programs), but which contain advice which is absolutely vital to successful real-world programming and which, if you can take it to heart early, will save you from a lot of misery out in the school of hard knocks later on.

Note that most of these notes merely amplify on the things K&R is saying; there isn’t much to say that it doesn’t already say, usually better. In particular, many of the things that I’ll comment on in the early chapters are discussed in more detail in the later chapters; by barging in with my know-it-all comments, I’m partially destroying the authors’ careful progression from an initial, slightly superficial overview to a more detailed, complete presentation. If these notes present more detail than you want to see at first, don’t worry (but please do let me know); just come back to them later to see if they clear up anything you’re still uncertain on. (Also, if you find the description in K&R adequately clear, you don’t have to read all of these notes, but do take note of the highlighted “deep sentences.”)

Preface

page ix: You’ll get some hint here that C has become a bit more formal as it has “grown up.” That formality is appropriate, and for the second edition of K&R to acknowledge it is appropriate, and for any
modern course in C programming to teach it is appropriate. Personally, I learned C before it had become quite so formalized, and occasionally my traditional biases will leak through. I’ll try to admit it when they do.

As the authors note, C is a relatively small language, but one which (to its admirers, anyway) wears well. C’s small, unambitious feature set is a real advantage: there’s less to learn; there isn’t excess baggage in the way when you don’t need it. It can also be a disadvantage: since it doesn’t do everything for you, there’s a lot you have to do yourself. (Actually, this is viewed by many as an additional advantage: anything the language doesn’t do for you, it doesn’t dictate to you, either, so you’re free to do that something however you want.)

Preface to the First Edition

page xi: This Preface, in a few spare paragraphs, sums up much of the philosophy of C and the authors’ philosophy about programming in general. Their comments in C’s size and scope are fundamental, and though no one may have fully recognized it at the time (or yet), this unassuming approach to the design of the language is surely a significant factor behind C’s success. I didn’t have the first paragraph of the Preface to the First Edition in front of me when I wrote my notes (just above) on the Preface to the Second Edition, but it’s not surprising that they’re similar.

As the authors say, they assume some familiarity with basic programming concepts; other notes in this series will give you a bit of help with those concepts if you need it. The authors also anticipate another theme of theirs, which is that they will stress learning by doing. (I’ll have more to say about this as the learning begins.)

Deep sentence: Besides showing how to make effective use of the language, we have tried where possible to illustrate useful algorithms and principles of good style and sound design.

The authors’ advice on style is good, and their design is sound. Pay attention to the things they say which go beyond the nuts-and-bolts details of C: there’s a lot to learn here about programming in general.

Introduction

page 2: Deep sentence: ...C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses.

C is sometimes referred to as a “high-level assembly language.” Some people think that’s an insult, but it’s actually a deliberate and significant aspect of the language. If you have programmed in assembly language, you’ll probably find C very natural and comfortable (although if you continue to focus too heavily on machine-level details, you’ll probably end up with unnecessarily non-portable programs). If you haven’t programmed in assembly language, you may be frustrated by C’s lack of certain higher-level features. In either case, you should understand why C was designed this way: so that seemingly-simple constructions expressed in C would not expand to arbitrarily expensive (in time or space) machine language constructions when compiled. If you write a C program simply and succinctly, it is likely to result in a succinct, efficient machine language executable. If you find that the executable resulting from a C program is not efficient, it’s probably because of something silly you did, not because of something
the compiler did behind your back which you have no control over. In any case, there’s no point in complaining about C’s low-level flavor: C is what it is.

Next we see a more detailed list of the things that are not “part of C.” It’s good to understand exactly what we mean by this. When we say that the C language proper does not do things like memory allocation or I/O, or even string manipulation, we obviously do not mean that there is no way to do these things in C. In fact, the usual functions for doing these things are specified by the ANSI C Standard with as much rigor as is the core language itself.

The fact that things like memory allocation and I/O are done through function calls has three implications:

1. the function calls to do memory allocation, I/O, etc. are no different from any other function calls;
2. the functions which do memory allocation, I/O, etc. do not know any more about the data they’re acting on than ordinary functions do (we’ll have more to say about this later); and
3. if you have specialized needs, you can do nonstandard memory allocation or I/O whenever you wish, by using your own functions and ignoring the standard ones provided.

The sentence that says “Most C implementations have included a reasonably standard collection of such functions” is historical; today, all implementations conforming to the ANSI C Standard have a very standard collection.

This aspect of C is very widely criticized; it is also used (justifiably) to argue that C is not a good teaching language. C aficionados love this aspect of C because it means that C does not try to protect them from themselves: when they know what they’re doing, even if it’s risky or obscure, they can do it. Students of C hate this aspect of C because it often seems as if the language is some kind of a conspiracy specifically designed to lead them into booby traps and ‘gotchas’.

This is another aspect of the language which it’s fairly pointless to complain about. If you take care and pay attention, you can avoid many of the pitfalls. These notes will point out many of the obvious (and not so obvious) trouble spots.

The last sentence of the Introduction is misleading: as we’ll see, it’s risky to defer to any particular compiler as a “final authority on the language.” A compiler is only a final authority on the language it accepts, and the language that a particular compiler accepts is not necessarily exactly C, no matter what the name of the compiler suggests. Most compilers accept extensions which are not part of standard C and which are not supported by other compilers; some compilers are deficient and fail to accept certain constructs which are in standard C. From time to time, you may have questions about what is truly standard and which neither you nor anyone you’ve talked to is able to answer. If you don’t have a copy of the standard (or if you do, but you discover that the standardese in which it’s written is impenetrable), you may have to temporarily accept the jurisdiction of your particular compiler, in order to get some program working today and under that particular compiler, but you’d do well to mark the code in question as suspect and the question in your head as “don’t know; still unanswered.”