PEARSON

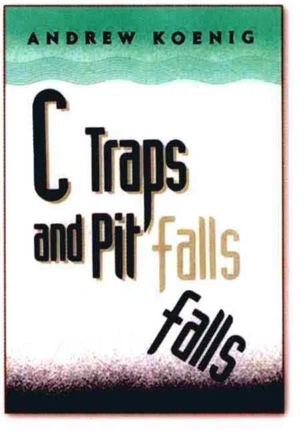
C和C++实务精选
品味岁月积淀，读享技术菁华

中文版
累计销量超
55 000册

C陷阱与缺陷

（英文版）

[美] Andrew Koenig 著



C Traps and Pitfalls

■ C语言经典著作

■ 集作者多年实际工作经验之大成

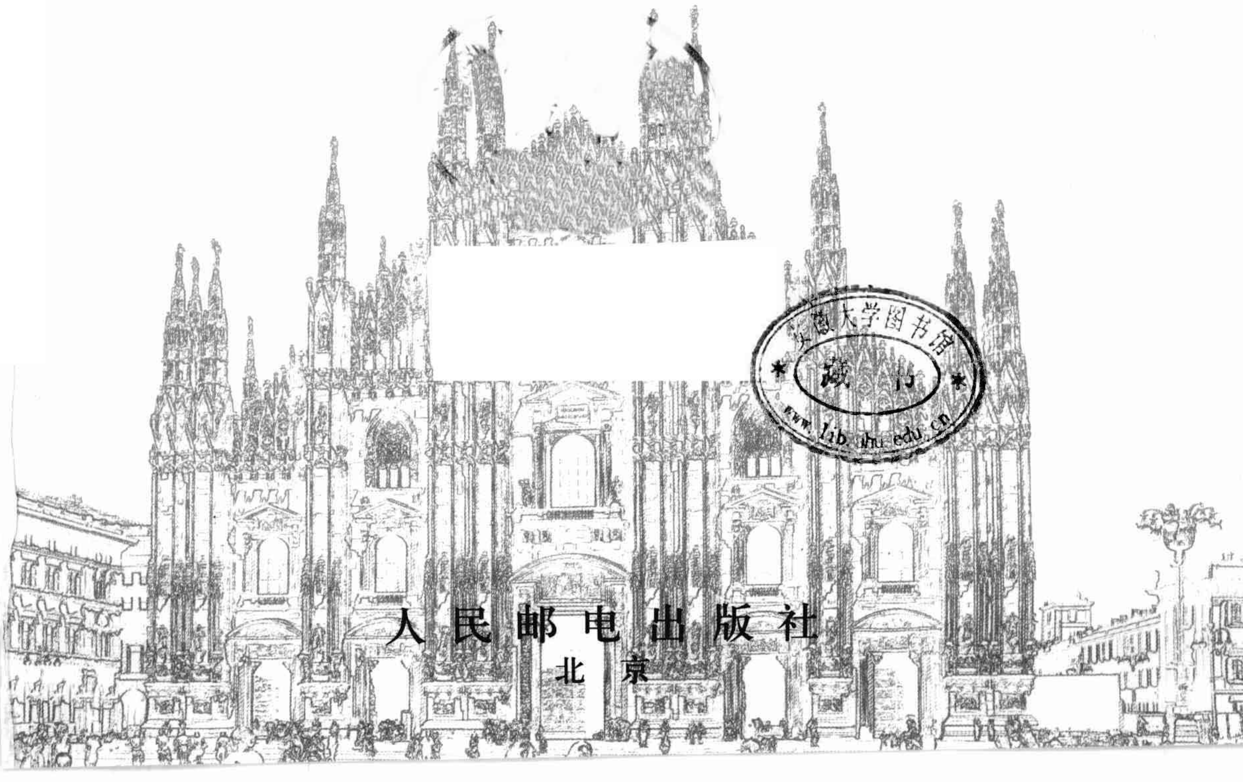
■ 帮助C程序员绕开经典陷阱和障碍

人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

C 陷阱与缺陷
（英文版）

[美] Andrew Koenig 著



人民邮电出版社
北京

图书在版编目（C I P）数据

C陷阱与缺陷 ： 英文 / （美） 凯尼格 （Koenig, A.） 著. -- 北京 ： 人民邮电出版社， 2013.2
ISBN 978-7-115-30859-7

Ⅰ. ①C… Ⅱ. ①凯… Ⅲ. ①C语言－程序设计－英文
Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第012337号

版权声明

C 陷阱与缺陷（英文版）

◆ 著 [美] Andrew koenig
责任编辑 汪 振

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：700×1000 1/16
印张：10
字数：207 千字 2013 年 2 月第 1 版
印数：1－2 500 册 2013 年 2 月北京第 1 次印刷

著作权合同登记号 图字：01-2012-9285 号

ISBN 978-7-115-30859-7

定价：39.00 元

读者服务热线：(010)67132692 印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

作者以 1985 年在 Bell 实验室时发表的一篇论文为基础，结合自己的工作经验，创作了这本对 C 程序员具有珍贵价值的经典著作。写作本书的出发点不是要批判 C 语言，而是要帮助 C 程序员绕过编程过程中的陷阱和障碍。

全书分为 8 章，分别从词法分析、语法语义、连接、库函数、预处理器、可移植性缺陷等几个方面分析了 C 编程中可能遇到的问题。最后，作者用一章的篇幅给出了若干具有实用价值的建议。

本书适合有一定经验的 C 程序员阅读学习，即便你是 C 编程高手，本书也应该成为你的案头必备书籍。

作者简介

Andrew Koenig

AT&T 大规模程序研发部（前贝尔实验室）成员。他从 1986 年开始从事 C 语言的研究，1977 年加入贝尔实验室。他编写了一些早期的类库，并在 1988 年组织召开了第一个相当规模的 C++会议。在 ISO/ANSI C++委员会成立的 1989 年，他就加入了该委员会，并一直担任项目编辑。他已经发表了 C++方面的 100 多篇论文，在 Addsion-Wesley 出版了 *C Trap and Pitfalls*,（《C 陷阱与缺陷》）和 *Ruminations on C++*（《C++沉思录》），还应邀到世界各地演讲。

Andrew Koenig 不仅有着多年的 C++开发、研究和教学经验，而且还亲身参与了 C++的演化和变革，对 C++的变化和发展起到重要的影响。

前言

对于经验丰富的行家而言，得心应手的工具在初学时的困难程度往往要超过那些容易上手的工具。刚刚接触飞机驾驶的学员，初航时总是谨小慎微，只敢沿着海岸线来回飞行，等他们稍有经验就会明白这样的飞行其实是一件多么轻松的事。初学骑自行车的新手，可能觉得后轮两侧的辅助轮很有帮助，但一旦熟练过后，就会发现它们很是碍手碍脚。

这种情况对程序设计语言也是一样。任何一种程序设计语言，总存在一些语言特性，很可能会给还没有完全熟悉它们的人带来麻烦。令人吃惊的是，这些特性虽然因程序设计语言的不同而异，但对于特定的一种语言，几乎每个程序员都在同样的一些特性上犯过错误、吃过苦头！因此，作者也就萌生了将这些程序员易犯错误的特性加以收集、整理的最初念头。

我第一次尝试收集这类问题是在 1977 年。当时，在华盛顿特区举行的一次 SHARE（IBM 大型机用户组）会议上，我作了一次题为“PL/I 中的问题与‘陷阱’”的发言。作此发言时，我刚从哥伦比亚大学调至 AT&T 的贝尔实验室，在哥伦比亚大学我们主要的开发语言是 PL/I，而贝尔实验室中主要的开发语言却是 C。在贝尔实验室工作的 10 年间，我积累了丰富的经验，深谙 C 程序员（也包括我本人）在开发时如果一知半解将会遇到多少麻烦。

1985 年，我开始收集有关 C 语言的此类问题，并在年底将结果整理后作为一篇内部论文发表。这篇论文所引发的回应却大大出乎我的意料，共有 2 000 多人向贝尔实验室的图书馆索取该论文的副本！我由此确信有必要将该论文的内容进一步扩充，于是就写成了现在读者所看到的这本书。

本书是什么

本书力图通过揭示一般程序员，甚至是经验老道的职业程序员，如何在编程中犯错误、摔跟头，以提倡和鼓励预防性的程序设计。这些错误实际上一旦被程序员

真正认识和理解，并不难避免。因此，本书阐述的重点不是一般原则，而是一个个具体的例子。

如果你是一个程序员并且开发中真正用到 C 语言来解决复杂问题，这本书应该成为你的案头必备书籍。即使你已经是一个 C 语言的专家级程序员，仍然有必要拥有这本书，很多读过本书早期手稿的专业 C 程序员常常感叹：“就在上星期我还遇到这样一个 Bug！”如果你正在教授 C 语言课程，本书毫无疑问应该成为你向学生推荐的首选补充阅读材料。

本书不是什么

本书不是对 C 语言的批评。程序员无论使用何种程序设计语言，都有可能遇到麻烦。本书浓缩了作者长达 10 年的 C 语言开发经验，集中阐述了 C 语言中各种问题和“陷阱”，目的是希望程序员读者能够从中吸取我本人以及我所见过的其他人所犯错误的经验教训。

本书不是一本“烹饪菜谱”。我们不能希望可以通过详尽的指导说明来完全避免错误。如果可行的话，那么所有的交通事故都可以通过在路旁刷上“小心驾驶”的标语来杜绝。对一般人而言最有效的学习方式是从感性的、活生生的事例中学习，比如自己的亲身经历或者他人的经验教训。而且，哪怕只是明白了一种特定的错误是如何可能发生的，就已经在将来避免该错误的路上迈了一大步。

本书并不打算教你如何用 C 语言编程（见 Kernighan 和 Ritchie：*The C Programming Language*，第 2 版，Prentice-Hall，1988），也不是一本 C 语言参考手册（见 Harbison 和 Steele：*C：A Reference Manual*，第 2 版，Prentice-Hall，1987）。本书未提及数据结构与算法（见 Van Wyk：*Data Structures And C Programs*，Addison-Wesley，1988），仅仅简略介绍了可移植性（见 Horton：*How To Write Portable Programs In C*，Prentice-Hall，1989）和操作系统接口（见 Kernighan 和 Pike：*The Unix Programming Environment*，Prentice-Hall，1984）。本书中所涉及的问题均来自编程实践，适当作了简化（如果希望读到一些“挖空心思”设计出来，专门让你绞尽脑汁的 C 语言难题，见 Feuer：*The C Puzzle Book*，Prentice-Hall，1982）。本书既不是一本字典也不是一本百科全书，我力图使其精简短小，以鼓励读者能够阅读全书。

读者的参与和贡献

可以肯定，我遗漏了某些值得注意的问题。如果你发现了一个 C 语言问题而本书又未提及，请通过 Addison-Wesley 出版社与我联系。在本书的下一版中，我很有可能引用你的发现，并且向你致谢。

关于 ANSI C

在我写作本书时，ANSI C 标准尚未最后定案。严格地说，在 ANSI 委员会完成其工作之前，“ANSI C”的提法从技术上而言是不正确的。而实际上，ANSI 标准化工作大体已经尘埃落定，本书中提及的有关 ANSI C 标准内容基本上不可能有所变动。很多 C 编译器甚至已经实现了大部分 ANSI 委员会所考虑的对 C 语言的许多重大改进。

毋需担心你使用的 C 编译器并不支持书中出现的 ANSI 标准函数语法，它并不会妨碍你理解例子中真正重要的内容，而且书中提及的程序员易犯错误其实与何种版本的 C 编译器并无太大关系。

致谢

本书中问题的收集整理工作绝非一人之力可以完成。以下诸位都向我指出过 C 语言中的特定问题，他们是 Steve Bellovin（6.3 节），Mark Brader（1.1 节），Luca Cardelli（4.4 节），Larry Cipriani（2.3 节），Guy Harris and Steve Johnson（2.2 节），Phil Karn（2.2 节），Dave Kristol（7.5 节），George W. Leach（1.1 节），Doug McIlroy（2.3 节），Barbara Moo（7.2 节），Rob Pike（1.1 节），Jim Reeds（3.6 节），Dennis Ritchie（2.2 节），Janet Sirkis（5.2 节），Richard Stevens（2.5 节），Bjarne Stroustrup（2.3 节），Ephraim Vishnaic（1.4 节），以及一位自愿要求隐去姓名者（2.3 节）。为简短起见，对于同一个问题此处仅仅列出了第一位向我指出该问题的人。我认为这些错误绝不是凭空臆造出来的，而且即使是，我想也没有人愿意承认。至少这些错误我本人几乎都犯过，而且有的还不止犯一次。

在书稿编辑方面许多有用的建议来自 Steve Bellovin，Jim Coplien，Marc Donner，Jon Forrest，Brian Kernighan，Doug McIlroy，Barbara Moo，Rob Murray，Bob Richton，Dennis Ritchie，Jonathan Shapiro，以及一些未透露姓名的审阅者。Lee McMahon 与 Ed Sitar 为我指出了早期手稿中的许多录入错误，使我避免了一旦成书后将要遇到的很多尴尬。Dave Prosser 为我指明了许多 ANSI C 中的细微之处。Brian Kernighan 提供了极有价值的排版工具和帮助。

与 Addison-Wesley 出版社合作是一件愉快的事情，感谢 Jim DeWolf, Mary Dyer，Lorraine Ferrier，Katherine Harutunian，Marshall Henrichs，Debbie Lafferty，Keith Wollman，和 Helen Wythe。当然，他们也从一些并不为我所知的人们那里得到了帮助，使本书最终得以出版，我在此也一并致谢。

前言

我需要特别感谢 AT&T 贝尔实验室的管理层，他们开明的态度和支持使我得以写作本书，包括 Steve Chappell，Bob Factor，Wayne Hunt，Rob Murray，Will Smith，Dan Stanzione 和 Eric Sumner。

本书书名受到 Robert Sheckley 的科幻小说选集的启发，其书名是 *The People Trap and Other Pitfalls*，*Snares*，*Devices and Delusions*（*as well as Two Sniggles and a Contrivance*）（1968 年由 Dell Books 出版）。

CONTENTS

CHAPTER 0: **INTRODUCTION**

I wrote my first computer program in 1966, in Fortran. I had intended it to compute and print the Fibonacci numbers up to 10,000: the elements of the sequence 1, 1, 2, 3, 5, 8, 13, 21, ..., with each number after the second being the sum of the two preceding ones. Of course it didn't work:

```
     I = 0
     J = 0
     K = 1
   1 PRINT 10, K
     I = J
     J = K
     K = I + J
     IF (K - 10000) 1, 1, 2
   2 CALL EXIT
  10 FORMAT (I10)
```

Fortran programmers will find it obvious that this program is missing an **END** statement. Once I added the **END** statement, though, the program still didn't compile, producing the mysterious message **ERROR 6**.

Careful reading of the manual eventually revealed the problem: the Fortran compiler I was using would not handle integer constants with more than four digits. Changing `10000` to `9999` solved the problem.

I wrote my first C program in 1977. Of course it didn't work:

```
#include <stdio.h>

main()
{
        printf("Hello world");
}
```

This program compiled on the first try. Its result was a little peculiar, though: the terminal output looked somewhat like this:

```
% cc prog.c
% a.out
Hello world%
```

Here the % character is the system's *prompt*, which is the string the system uses to tell me it is my turn to type. The % appears immediately after the `Hello world` message because I forgot to tell the system to begin a new line afterwards. Section 3.10 (page 51) discusses an even subtler error in this program.

There is a real difference between these two kinds of problem. The Fortran example contained two errors, but the implementation was good enough to point them out. The C program was technically correct – from the machine's viewpoint it contained no errors. Hence there were no diagnostic messages. The machine did exactly what I told it; it just didn't do quite what I had in mind.

This book concentrates on the second kind of problem: programs that don't do what the programmer might have expected. More than that, it will concentrate on ways to slip up that are peculiar to C. For example, consider this program fragment to initialize an integer array with `N` elements:

```
int i;
int a[N];
for (i = 0; i <= N; i++)
        a[i] = 0;
```

On many C implementations, this program will go into an infinite loop! Section 3.6 (page 36) shows why.

Programming errors represent places where a program departs from the programmer's mental model of that program. By their very nature they are thus hard to classify. I have tried to group them according to their relevance to various ways of looking at a program.

At a low level, a program is as a sequence of *symbols*, or *tokens*, just as a book is a sequence of words. The process of separating a program into symbols is called *lexical analysis*. Chapter 1 looks at problems that stem from the way C lexical analysis is done.

One can view the tokens that make up a program as a sequence of statements and declarations, just as one can view a book as a collection of sentences. In both cases, the meaning comes from the details of how tokens or words are combined into larger units. Chapter 2 treats errors that can arise from misunderstanding these *syntactic* details.

Chapter 3 deals with misconceptions of meaning: ways a programmer who intended to say one thing can actually be saying something else. We assume here that the lexical and syntactic details of the language are well understood and concentrate on *semantic* details.

Chapter 4 recognizes that a C program is often made out of several parts that are compiled separately and later bound together. This process is called *linkage* and is part of the relationship between the program and its environment.

That environment includes some set of *library routines*. Although not strictly part of the language, library routines are essential to any C program that does anything useful. In particular, a few library routines are used by almost every C program, and there are enough ways to go wrong using them to merit the discussion in Chapter 5.

Chapter 6 notes that the program we write is not really the program we run; the preprocessor has gotten at it first. Although various preprocessor implementations differ somewhat, we can say useful things about aspects that many implementations have in common.

Chapter 7 discusses portability problems – reasons a program might run on one implementation and not another. It is surprisingly hard to do even simple things like integer arithmetic correctly.

Chapter 8 offers advice in defensive programming and answers the exercises from the other chapters.

Finally, an Appendix covers three common but widely misunderstood library facilities.

Exercise 0-1. Would you buy an automobile made by a company with a high proportion of recalls? Would that change if they told you they had cleaned up their act? What does it *really* cost for your users to find your bugs for you? □

Exercise 0-2. How many fence posts 10 feet apart do you need to support 100 feet of fence? □

Exercise 0-3. Have you ever cut yourself with a knife while cooking? How could cooking knives be made safer? Would you want to use a knife that had been modified that way? □

CHAPTER 1: **LEXICAL PITFALLS**

When we read a sentence, we do not usually think about the meaning of the individual letters of the words that make it up. Indeed, letters mean little by themselves: we group them into words and assign meanings to those words.

So it is also with programs in C and other languages. The individual characters of the program do not mean anything in isolation but only in context. Thus in

```
	p->s = "->";
```

the two instances of the - character mean two different things. More precisely, each instance of - is part of a different *token:* the first is part of -> and the second is part of a character string. Moreover, the -> token has a meaning quite distinct from that of either of the characters that make it up.

The word *token* refers to a part of a program that plays much the same role as a word in a sentence: in some sense it means the same thing every time it appears. The same sequence of characters can belong to one token in one context and an entirely different token in another context. The part of a compiler that breaks a program up into tokens is often called a *lexical analyzer*.

For another example, consider the statement:

```
	if (x > big) big = x;
```

The first token in this statement is `if`, a keyword. The next token is the left parenthesis, followed by the identifier `x`, the "greater than" symbol, the identifier `big`, and so on. In C, we can always insert extra space (blanks, tabs, or newlines) between tokens, so we could have written: