TURING 图灵程序设计丛书

Microsoft

Writing Solid Code

编程精粹

编写高质量C语言代码

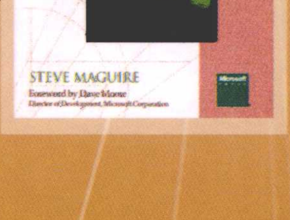
（英文版）

[美] Stephen A. Maguire 著

JOLT PRODUCTIVITY AWARD

Jolt生产效率大奖得主

- 与《代码大全》齐名的经典著作
- 揭示微软成功的技术奥秘
- C语言高手的秘籍



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Writing Solid Code

编程精粹

编写高质量C语言代码

（英文版）

[美] Stephen A. Maguire 著

人民邮电出版社

北 京

图书在版编目（CIP）数据

编程精粹：编写高质量 C 语言代码 = Writing Solid Code：英文 /（美）马圭尔（Maguire, S. A.）著．—北京：人民邮电出版社，2009.2
ISBN 978-7-115-19316-2

Ⅰ.编… Ⅱ.马… Ⅲ.C 语言－程序设计－英文 Ⅳ.TP312

中国版本图书馆CIP数据核字（2008）第191154号

内 容 提 要

软件日趋复杂，编码错误随之而来。要在测试前发现程序的错误，开发出无错误的程序，关键是弄清楚错误为何产生，又是如何产生。本书给出了多条编程方面的指导，这些指导看似简单，却是作者多年思考及实践的结果，是对其编程经验的总结。书中解决问题的思考过程对于程序开发人员尤显珍贵。

本书适于各层次程序开发人员阅读。

图灵程序设计丛书

编程精粹：编写高质量C语言代码（英文版）

◆ 著　　[美] Stephen A. Maguire
责任编辑　傅志红

◆ 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　　电子函件　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：800×1000　1/16
印张：17.75
字数：341千字　　　2009年2月第1版
印数：1－3 000册　　　2009年2月北京第1次印刷
著作权合同登记号　图字：01-2008-5502号

ISBN 978-7-115-19316-2/TP

定价：45.00元

读者服务热线：(010)88593802　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

To my wife, Beth,
and to my parents, Joseph and Julia Maguire,
for all their love and support.

FOREWORD

I first met Steve Maguire in 1986, when we hired him to work on Macintosh Excel. He impressed me then as a particularly conscientious and dedicated programmer. At that time, I was the development manager for Microsoft Multiplan, Word, and Chart. The company was growing rapidly, and so were problems with both our products and our development process. Steve was instrumental in solving some of those problems and with this book becomes the recorder of many good practices we developed in response to those problems. But I'm getting ahead of myself.

I was hired by Bill Gates and Charles Simonyi in 1981 to work in Microsoft's business applications group. Back then, that meant 7 programmers working on one business application—Microsoft Multiplan. Another 30 programmers were working on our language and operating systems products. The rest of the 100 people in the company were in technical writing, sales, marketing, and administration. At that time, all 7 Multiplan programmers were crammed into one large room in an office building in downtown Bellevue, Washington. We weren't even in the same building with the rest of the developers, who were working on MS-DOS and Basic. They were two blocks away. But that wasn't a big problem. We were a small company with a vision of what we wanted to accomplish: a computer on every desk running Microsoft software.

The system we used to develop Multiplan was pretty sophisticated for PC development in those days. We wrote the core product in C—most programs then were written in assembly or Pascal. We did our editing and compilation on a PDP-11 running Unix. The C code was compiled into p-code and downloaded to the target machines. We had to build p-code interpreters for each microprocessor in use at that time.

By the end of 1983, we had interpreters working for the 8080/Z80, the 6502, the Z8000, the 68000, the TI 99/a, and the 8086. And by that time, we had application specialists working on each of our primary business applications—a spreadsheet, a word processor, a simple database record manager, and a business graphics package. We had assembly language and environment specialists working on the interpreters. We also had a group working on the compiler and development tools. Except for a small dependence on the minuscule operating system services, the 30-member application development team was self-contained, building its own development tools, compilers, interpreters, and product code.

In 1981, our primary focus had been on shipping original equipment manufacturer products. We would work with an OEM, customizing our products to fit the OEM's machine and sales channels. Then we would ship the OEM a disk and photo-ready copies of the manual. The OEM would do all of the manufacturing of the product, the sales, and the support.

By 1982, we had started to switch to a retail emphasis. The OEM focus had allowed us to travel light. We'd needed only a few marketing folks to sell the products to the OEMs, a few developers to build the products, and a few technical writers to write the manuals. Testing, project management, product manufacturing, product shipping, product support, and sales had been provided by the OEM. With the switch to a retail focus, we had to develop all of these specialized product development and support functions at Microsoft.

Early on, we developed products for IBM and Apple PCs. Our first retail products were Multiplan for IBM-DOS and Multiplan for the Apple II. But we still developed many OEM products. We worked on spreadsheet, word processing, business graphics, and database products for Unix, Xenix, the TI 99/a, the Tandy M100, the MSX (an 8-bit home computer in Japan), non-IBM-compatible MS-DOS machines, the Commodore 64, the Atari, the Apple III, the Apple Lisa, the Apple Macintosh, OS/2, Windows, and many other specialized hardware environments. Some of these environments had several variants themselves. Before the IBM-compatible became the dominant machine, we'd had to tailor our applications for every MS-DOS machine that was built. We'd had a different product for the Tandy, the Wang, the Paradyne, the Consumer Devices, the Eagle, the Victor, the Olivetti, the DEC Rainbow, and many other MS-DOS machines. While dealing with this system specialization, we were developing numerous specialized foreign language versions of our business applications.

Our early products were only English language versions. Today we build over 30 language products that we adapt, or more often tailor, to the target language/culture, including Arabic, Australian, Bahas, Chinese, Czechoslovakian, Danish, Dutch, English (UK), Finnish, French, French

Canadian, German, Greek, Hebrew, Honguel (Korean), Italian, Japanese, Norwegian, Portuguese, Russian, Spanish, Swedish, Turkish, US English, and more.

By 1985, some of the complexity of product development had been eliminated by the success of the IBM PC. The variety of video standards we'd had to support had been reduced to the primary IBM-compatible modes (CGA and monochrome). But video support started to get out of hand again around 1988. IBM had developed the EGA video extensions, then they developed the VGA, and it was soon followed by the SVGA and all of its variants.

Support for the other hardware peripherals also grew more complex. We had to support over 200 variations of laser and dot matrix printers. Fortunately, input devices didn't get too varied. There was the IBM standard keyboard and the extended keyboard. And most pointing devices followed the Microsoft mouse standard.

Today a lot of the complexity and variations in the hardware have simply gone away or have been incorporated into one complex but complete interface. We have to build products for only two primary systems—Windows and the Mac. But new levels and magnitudes of complexity have emerged to replace the complexities of hardware support. Now developers need to be conversant with message-based GUI programming and with object-oriented design and programming. They need to support product extensibility through Object Linking and Embedding (OLE) in Windows and through Publish and Subscribe on the Mac. And they need to support consistent access to features across product families and consistent methods of programmability across product families.

In 1984, the increase in the complexity of our products and the high standards involved in building retail products led us to start up a quality assurance group. We called this group Testing in 1984, and we call this group Testing today, although our testing group has grown from 5 testers in 1984 to over 500 testers. Our testing group today is really an advanced quality assurance group that looks out for our customer's interests.

Before we'd had a testing group, the business applications developers had relied on the OEM customer to test the product to find bugs. This arrangement worked out well until we started to ship the retail product directly to end users, before we'd shipped it to any OEM customers. For an interval, before the testing group was going strong, the developers had to test the retail products themselves. The developers who lived through that experience learned that they had to be very careful not to introduce bugs as they wrote and debugged the code. They found out the hard way how costly it was to release a product that had bugs in it.

But as the testing group got bigger, the development groups got more and more dependent on the testing group to find bugs. The development groups soon adopted the attitude that the testing group was responsible for finding all bugs. This led to such serious problems—slipped schedules, buggy features, incomplete features, even canceled products—that something had to be done. Many developers felt no shame if bugs were found in their code after the product had shipped. They'd ask indignantly, "Why didn't Testing find that bug before we shipped?" Testing should have responded, "Why did you put that bug in the product in the first place?"

Eventually, the developers began to realize that Testing can never find all of the bugs in a piece of software. The bugs might be in the design, or in the specifications, or in the analysis of the customer's needs. And testers can't do complete code coverage or path coverage in their tests. Bugs might be hidden in obscure and rarely tested code. Bugs can be temporarily masked by the operations of other parts of the program—or by the testing environment. These are the kinds of bugs that testers have a hard time finding. Because of these factors, a testing group will usually find only 60 percent of the bugs in a product.

The developers can bring more knowledge and tools to reviewing and testing the code. When the developers set their minds and their tools to it, they can find over 90 percent of the bugs in the code. If the developers give the responsibility for finding the bugs to the testers, the users of the product will find 40 percent of the bugs. If Development and Testing both work to find the bugs, the users will end up finding less than 4 percent of the bugs. And that 4 percent could be found by the users during the beta test of the product.

In early 1989, many of the development managers and leads met to discuss the problem. Out of that meeting came a realization and an attitude change: Finding and fixing bugs was Development's responsibility. Development had been letting bugs slip past them. Now it became their responsibility again to prevent bugs from being released to Testing and then on to the customers. The development teams set off on the goal of having a "nearly shippable product every day." This means that when a feature is marked complete, any bugs found in it will have to be fixed before any new work is attempted. Work in progress will be brought to a standstill if serious bugs are found in features marked complete.

We labeled this new attitude "zero defects." The code would be built, reviewed, and tested by Development and delivered to Testing with zero defects. Fortunately, a few of the development groups had already been experimenting with many of the techniques for developing zero defect code. We started to actively share those techniques among all the development groups. Steve Maguire did a lot of troubleshooting from group to group in those days, and he has set down many of our techniques for writing solid, bug-free code in this book.

Microsoft improved and is always improving its product development process along with its development tools. In 1981, there were the developers, the writers of the manuals, and small sales, marketing, and administrative groups. Now we have product marketing, channel marketing, sales, support, testing, user education (technical writing and publishing), program management, and many other specialists. With today's complex structure of special groups at Microsoft, we want to ensure that the techniques for developing solid code aren't lost, misunderstood, or forgotten. Steve Maguire's book should help both us and you keep those techniques alive.

Today I'm the director of development and testing for Microsoft. Part of my job is to inventory and disseminate best practices. I'm very grateful to Steve for taking the time to write a book so enjoyable to read that will help managers and programmers develop world-class code. Steve has captured and described many of the techniques that are used at Microsoft to develop solid, shippable code. It will become recommended reading for all Microsoft programmers.

David M. Moore
Director of Development, Microsoft
Redmond, Washington
January 1993

PREFACE

In 1986, after 10 years of consulting and working for small companies, I went to work for Microsoft specifically to get experience in writing Macintosh applications. I joined Microsoft's Excel team, the group responsible for the company's graphical spreadsheet application.

I'm not sure what I was expecting the code to look like—glamorous or elegant, I suppose. What I found was plain, everyday code, nothing much different from what I'd seen before. To be sure, the spreadsheet had a wonderful user interface—it was much easier and more intuitive to use than any of the character-based spreadsheets of the time. But what impressed me even more was the implementation of an extensive debugging system built into the product.

The system automatically alerted programmers and testers to bugs, much the way warning lights in the cockpit of a Boeing 747 alert pilots to failures—the debugging system was not so much testing the code as it was *monitoring* it. None of the concepts in the debugging system were new, but I was struck by the sheer extent to which they were employed, and by how effective the system was in detecting bugs. It was an eye-opener. It didn't take me long to discover that most of Microsoft's projects had extensive internal debugging systems—and that there was a heightened awareness among the programmers of bugs and their causes.

I worked on Macintosh Excel for two years before I left to help another Microsoft group, whose code was turning up with a higher than usual number of bugs. I found that during the two years in which I had been focused on Excel, Microsoft had tripled in size and many of the programming concepts that were well-known among the older groups had not spread to the newer groups during the rapid growth. Instead of having a heightened awareness of error-prone coding practices, the newer programmers had a normal awareness—about what I'd seen among programmers in the years before I joined Microsoft.

About six months after I'd moved to the new group, I was talking to a fellow programmer and mentioned that somebody should document the concepts behind writing bug-free code so that the principles could spread to the newer groups. The other programmer looked at me and said, "You don't seem to mind writing documents; why don't *you* write down the details? In fact, why don't you write a book and see if Microsoft Press will publish it? After all, none of this information is proprietary; it simply makes programmers more aware of bugs."

I didn't give that suggestion much thought then, mainly because I didn't have the time and I'd never written a book before—the closest I'd come to authorship was cowriting a programming column for *Hi-Res Magazine* in the early 1980s. Not quite the same thing.

But as you can see, the book did get written, and for a simple reason: In 1989 Microsoft canceled an unannounced product because of a runaway bug list. Now, runaway bug lists weren't new—several of Microsoft's competitors had already canceled projects because of them. But this was the first time that Microsoft had ever canceled a project for that reason. It was also the latest in a string of buggy products, and management had finally said, "Enough is enough" and taken a series of steps to get bug counts back down to their previous levels. Still, nobody was given responsibility for putting the details down on paper.

By this time the company was nine times larger than when I'd started, and I didn't see how the company's coding could return to its previous low bug levels without explicit, recorded guidelines, particularly when I considered the growing complexity of Windows and Macintosh applications. That's when I decided, finally, to write this book.

Microsoft Press agreed to publish it.

And here it is.

I hope you enjoy reading the book. I've tried to keep it informal and entertaining.

ACKNOWLEDGMENTS

I'd like to thank everybody at Microsoft Press who helped make this book a reality, and in particular the two people who held my hand throughout the writing process. First I would like to thank Mike Halvorson, my acquisitions editor, for letting me take the project at my own speed and for patiently answering this first-time book author's many questions. I would especially like to thank Erin O'Connor, my manuscript editor, who gave me early feedback on the chapters, and without whose help this book simply would not exist. Erin also encouraged me to relax into my own style, and it certainly didn't hurt that she laughed at the text's little jokes. Jeff Carey gave the ideas and the code a good going over, and Kathleen Atkins made many good suggestions.

I'd also like to thank my father, Joseph Maguire, who in the mid-1970s introduced me to those first microcomputers: the Altair, the IMSAI, and the Sol-20. He is responsible for getting me hooked on this business. Evan Rosen, with whom I worked at Valpar International from 1981 to 1983, was a great influence on me, and his knowledge and insight show up in this book. Paul Davis, with whom I've had the pleasure to work during the past 10 years on various projects all over the country, has also shaped my thinking in significant ways.

I'd like to thank all the people who took the time to read through draft copies of this book to give me technical feedback: Mark Gerber, Melissa Glerum, Chris Mason, Dave Moore, John Rae-Grant, and Alex Tilles. I'd especially like to thank Eric Schlegel and Paul Davis for not only reviewing draft copies of the book but also giving me early help in hammering out the details.

Seattle, Washington
October 22, 1992

Introduction

Several years ago I picked up a copy of *TEX: The Program,* by Donald Knuth, and what I read in the preface astounded me:

> I believe that the final bug in TEX was discovered and removed on November 27, 1985. But if, somehow, an error still lurks in the code, I shall gladly pay a finder's fee of $20.48 to the first person who discovers it. (This is twice the previous amount, and I plan to double it again in a year; you see, I really am confident!)

I have no idea whether Knuth paid anybody $20.48 or even $40.96; that's not important. What *is* important is the confidence Knuth had in the quality of his code. How many programmers do you know who would seriously claim that their programs are totally bug-free? How many would publish such a claim and back it up with a finder's fee?

Programmers could make such claims if they truly believed that their testing groups had found all their bugs. But that's the problem. How many times have you heard programmers say, "I hope Testing has found all the bugs" right before the code is boxed, shrink-wrapped, and shipped to dealers? They cross their fingers and hope for the best.

Programmers today aren't sure their code is bug-free because they've relinquished responsibility for thoroughly testing it. It's not that management ever came out and said, "Don't worry about testing your code—the testers will do that for you." It's more subtle than that. Management expects programmers to test their code, but they expect testers to be more thorough; after all, that's Testing's full-time job.

The purpose of this book is to show how programmers can take back the responsibility for writing bug-free code. That doesn't necessarily mean writing perfect code the first time—it means creating a product that's bug-free before it first goes into testing. Some programmers may laugh incredulously at such an idea, but this book demonstrates techniques and provides guidelines that programmers can use to work toward that goal.

THE TWO CRITICAL QUESTIONS

The most critical requirement for writing bug-free code is to become attuned to what causes bugs. All of the techniques and guidelines presented in this book are the result of programmers asking themselves two questions over and over again, year after year, for every bug found in their code:

- How could I have *automatically* detected this bug?
- How could I have *prevented* this bug?

The easy answer to both questions would be "better testing," but that's not automatic, nor is it really preventive. Answers like "better testing" are so general they have no muscle—they're effectively worthless. Good answers to these questions result in specific techniques that eliminate the kind of bug you've just found.

This book is devoted to techniques and guidelines that have been found to reduce or completely eliminate entire classes of bugs. Some of its points smack right up against common coding practices, but before dismissing them with "everybody breaks that guideline," or "nobody does that," stop and think it through for yourself. If "nobody does that," why not? Make sure the reasons are still valid. Practices that made sense when FORTRAN was the hot new language may not make sense now.

That's not to say that you should blindly follow the guidelines in this book. They aren't rules. Too many programmers have taken the guideline "Don't use *goto* statements" as a commandment from God that should never be broken. When asked why they're so strongly against *goto*s, they say that using *goto* statements results in unmaintainable spaghetti code. Experienced programmers often add that *goto* statements can upset the compiler's code optimizer. Both points are valid. Yet there are times when the judicious use of a *goto* can greatly improve the clarity and efficiency of the code. In such cases, clinging to the guideline "Don't use *goto* statements" would result in worse code, not better.

The guidelines in this book are no different: They're meant to be followed most of the time, and they're meant to be broken when you can get better results by breaking them.

In addition to the guidelines and techniques, most of the chapters in this book contain a section at the end called "Things to Think About." Questions in this section of a chapter explore new areas that haven't been covered in the earlier parts of the chapter. The questions aren't exercises—they don't test your comprehension of the chapter. I've tried to introduce at least one new concept in every question, and I've provided a complete set of answers in order to pass on as much information as possible. If you usually skip over exercises, consider reading the answers in Appendix C so that you won't miss any of the guidelines or techniques I've introduced there.

Building atop Existing Foundations

Programmers who have been using C for a while know that they should use parentheses around arguments in macro definitions; they know that strings have unseen nul characters; they know that C arrays start with element *0*, not *1*; and they know that you must use *break* statements to prevent *switch* cases from falling into each other. These and other misunderstandings about the C language are common sources of bugs, but you won't find these bugs under discussion in this book unless such discussion is part of another point I'm making. I have tried to focus on the little-known, or rarely published, techniques for writing bug-free code, techniques that you won't usually find in programming textbooks or hear about in programming courses.

Nor have I tried to rehash guidelines already covered so well in *The Elements of Programming Style*, the programming classic written by Brian Kernighan and P. J. Plauger. Although Kernighan and Plauger use FORTRAN and PL/I in their examples, their guidelines—with a few exceptions—are applicable to any programming language, including C. *Writing Solid Code* builds on the groundwork laid by *The Elements of Programming Style* and follows a similar format.

Finally, although this book is written for professional programmers working on real projects with real deadlines, it's also suitable for students in advanced C programming courses. Few students will ever work on a compiler once they finish their compiler course, but all will have to focus on writing bug-free code. It's my hope that this book will help give students the skills they'll need to write solid, production quality code once they graduate.

WHAT'S A "MACINTOSH"?

Sometimes it almost seems that a book won't be taken seriously unless it mentions the PDP-11, the IBM 360, or some other old piece of hardware. So there, I've mentioned them, and I won't mention them again in this book. The systems you will hear a lot about in this book are MS-DOS, Microsoft Windows, and especially the Apple Macintosh—because those are the systems I've written code for most recently.

You'll also hear a lot about the history of the Microsoft Excel and Microsoft Word applications in this book. Excel is Microsoft's graphical spreadsheet, originally written for the Macintosh and later significantly rewritten, cleaned up, and enhanced for Windows.

Throughout the book, I talk about my experiences as a Macintosh Excel programmer, but I must confess that I spent most of my time either porting Windows code to the Macintosh sources or implementing look-alike features that Windows Excel already had. I had little to do with the phenomenal success of the product.

My only strategic contribution to Macintosh Excel was to convince Microsoft to kill it, and to instead build the Macintosh version directly from the much-improved Windows version's sources. Macintosh Excel 2.2 was the first version based on Windows Excel, sharing 80% of the code with its sibling. This was great for Macintosh Excel users because with the 2.2 release they saw a big jump in features and quality.

Word is Microsoft's word processing application. Actually, there are three versions of Word: Word for MS-DOS, which is character-based; Word for the Macintosh; and Word for Windows. As I write, the three products are still built from separate sources, but the versions are similar enough that most users can move among them without much difficulty. Eventually, all versions of Word will be built from common sources. The work is in progress.

WHAT ABOUT THE CODE?

You don't need to be an MS-DOS, Microsoft Windows, or Apple Macintosh expert to follow the book's code—the code is written in straightforward C that should compile and run with any ANSI C development system.

However, if you're a mainframe or minicomputer programmer without much experience on microcomputers, be aware that protected memory support is still rare in microcomputer operating systems. You can read and