HZ BOOKS

PEARSON Addison Wesley

Modern C++ Design C++ 设计新思维

More Effective C++

35 New ways to Improve Your Programs and Designs

（英文版）

（美） Scott Meyers 著

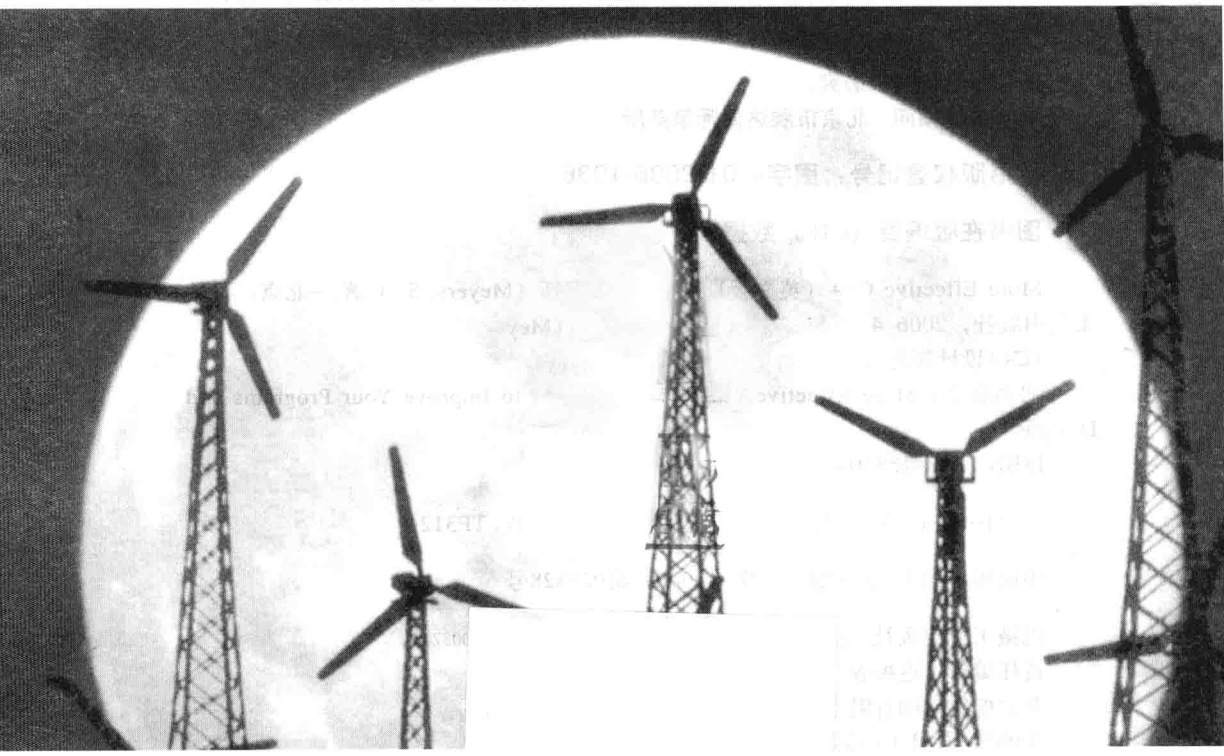
机械工业出版社
China Machine Press

More Effective C++

(英文版)

35 New ways to Improve Your Programs and Designs

(美) Scott Meyers 著



机械工业出版社
China Machine Press

本书版权登记号：图字：01-2006-1936

图书在版编目（CIP）数据

More Effective C++（英文版）/（美）迈耶斯（Meyers, S.）著. -北京：机械工业出版社，2006.4

（C++设计新思维）

书名原文：More Effective C++: 35 New Ways to Improve Your Programs and Designs

ISBN 7-111-18830-6

Ⅰ.M…　Ⅱ.迈…　Ⅲ.C语言-程序设计-英文　Ⅳ.TP312

中国版本图书馆CIP数据核字（2006）第029828号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：迟振春

北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行

2006年4月第1版第1次印刷

718mm × 1020mm 1/16 · 21印张

定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

“C++设计新思维”丛书前言

自C++诞生尤其是ISO/ANSI C++标准问世以来，以Bjarne Stroustrup为首的C++社群领袖一直不遗余力地倡导采用“新风格”教学和使用C++。事实证明，除了兼容于C的低阶特性外，C++提供的高级特性以及在此基础上发展的各种惯用法可以让我们编写出更加简洁、优雅、高效、健壮的程序。

这些高级特性和惯用法包括精致且高效的标准库和各种“准标准库”，与效率、健壮性、异常安全等主题有关的各种惯用法，以及在C++的未来占据更重要地位的模板和泛型程序设计技术等。它们发展于力量强大的C++社群，并被这个社群中最负声望的专家提炼、升华成一本本精彩的著作。毫无疑问，这些学术成果必将促进C++社群创造出更多的实践成果。

我个人认为，包括操作系统、设备驱动、编译器、系统工具、图像处理、数据库系统以及通用办公软件等在内的基础软件更能够代表一个国家的软件产业发展质量，迄今为止，此类基础性的软件恰好是C++所擅长开发的，因此，可以感性地说，C++的应用水平在一定程度上可以折射出一个国家的软件产业发展水平和健康程度。

前些年国内曾引进出版了一大批优秀的C++书籍，它们拓宽了中国C++程序员的视野，并在很大程度上纠正了长期以来存在于C++的教育、学习和使用方面的种种误解，对C++相关的产业发展起到了一定的促进作用。然而在过去的两年中，随着.NET、Java技术吸引越来越多的注意力，中国软件产业业务化、项目化的状况愈发加剧，擅长于“系统编程”的C++语言的应用领域似有进一步缩减的趋势，这也导致人们对C++的出版教育工作失去了应有的重视。

机械工业出版社华章分社决定继续为中国C++“现代化”教育推波助澜，从2006年起将陆续推出一套“C++设计新思维”丛书。这套丛书秉持精品、高端的理念，其作译者为包括Herb Sutter在内的国内外知名C++技术专家和研究者、教育者，议题紧密围绕现代C++特性，以实用性为主，兼顾实验性和探索性，形式上则是原版影印、中文译著和原创兼收并蓄。每一本书相对独立且交叉引用，篇幅短小却内容深入。作为这套丛书的特邀技术编辑，我衷心希望它们所展示的技术、技巧和理念能够为中国C++社群注入新的活力。

荣 耀
2005年12月
南京师范大学
www.royaloo.com

Praise for *More Effective C++: 35 New Ways to Improve Your Programs and Designs*

"This is an enlightening book on many aspects of C++: both the regions of the language you seldom visit, and the familiar ones you THOUGHT you understood. Only by understanding deeply how the C++ compiler interprets your code can you hope to write robust software using this language. This book is an invaluable resource for gaining that level of understanding. After reading this book, I feel like I've been through a code review with a master C++ programmer, and picked up many of his most valuable insights."

— Fred Wild, Vice President of Technology,
Advantage Software Technologies

"This book includes a great collection of important techniques for writing programs that use C++ well. It explains how to design and implement the ideas, and what hidden pitfalls lurk in some obvious alternative designs. It also includes clear explanations of features recently added to C++. Anyone who wants to use these new features will want a copy of this book close at hand for ready reference."

— Christopher J. Van Wyk, Professor,
Mathematics and Computer Science, Drew University

"Industrial strength C++ at its best. The perfect companion to those who have read Effective C++*."*

— Eric Nagler, C++ Instructor and Author,
University of California Santa Cruz Extension

"More Effective C++ *is a thorough and valuable follow-up to Scott's first book,* Effective C++. *I believe that every professional C++ developer should read and commit to memory the tips in both* Effective C++ *and* More Effective C++. *I've found that the tips cover poorly understood, yet important and sometimes arcane facets of the language. I strongly recommend this book, along with his first, to developers, testers, and managers ... everyone can benefit from his expert knowledge and excellent presentation."*

— Steve Burkett, Software Consultant

Acknowledgments

A great number of people helped bring this book into existence. Some contributed ideas for technical topics, some helped with the process of producing the book, and some just made life more fun while I was working on it.

When the number of contributors to a book is large, it is not uncommon to dispense with individual acknowledgments in favor of a generic "Contributors to this book are too numerous to mention." I prefer to follow the expansive lead of John L. Hennessy and David A. Patterson in *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, first edition 1990). In addition to motivating the comprehensive acknowledgments that follow, their book provides hard data for the 90-10 rule, which I refer to in Item 16.

The Items

With the exception of direct quotations, all the words in this book are mine. However, many of the ideas I discuss came from others. I have done my best to keep track of who contributed what, but I know I have included information from sources I now fail to recall, foremost among them many posters to the Usenet newsgroups `comp.lang.c++` and `comp.std.c++`.

Many ideas in the C++ community have been developed independently by many people. In what follows, I note only where *I* was exposed to particular ideas, not necessarily where those ideas originated.

Brian Kernighan suggested the use of macros to approximate the syntax of the new C++ casting operators I describe in Item 2.

In Item 3, my warning about deleting an array of derived class objects through a base class pointer is based on material in Dan Saks' "Gotchas" talk, which he's given at several conferences and trade shows.

In Item 5, the proxy class technique for preventing unwanted application of single-argument constructors is based on material in Andrew Koenig's column in the January 1994 *C++ Report*.

James Kanze made a posting to `comp.lang.c++` on implementing postfix increment and decrement operators via the corresponding prefix functions; I use his technique in Item 6.

David Cok, writing me about material I covered in *Effective C++*, brought to my attention the distinction between `operator new` and the `new` operator that is the crux of Item 8. Even after reading his letter, I didn't really understand the distinction, but without his initial prodding, I probably *still* wouldn't.

The notion of using destructors to prevent resource leaks (used in Item 9) comes from section 15.3 of Margaret A. Ellis' and Bjarne Stroustrup's *The Annotated C++ Reference Manual* (see page 285). There the technique is called *resource acquisition is initialization*. Tom Cargill suggested I shift the focus of the approach from resource acquisition to resource release.

Some of my discussion in Item 11 was inspired by material in Chapter 4 of *Taligent's Guide to Designing Programs* (Addison-Wesley, 1994).

My description of over-eager memory allocation for the `DynArray` class in Item 18 is based on Tom Cargill's article, "A Dynamic vector is harder than it looks," in the June 1992 *C++ Report*. A more sophisticated design for a dynamic array class can be found in Cargill's follow-up column in the January 1994 *C++ Report*.

Item 21 was inspired by Brian Kernighan's paper, "An AWK to C++ Translator," at the 1991 USENIX C++ Conference. His use of overloaded operators (sixty-seven of them!) to handle mixed-type arithmetic operations, though designed to solve a problem unrelated to the one I explore in Item 21, led me to consider multiple overloadings as a solution to the problem of temporary creation.

In Item 26, my design of a template class for counting objects is based on a posting to `comp.lang.c++` by Jamshid Afshar.

The idea of a mixin class to keep track of pointers from `operator new` (see Item 27) is based on a suggestion by Don Box. Steve Clamage made the idea practical by explaining how `dynamic_cast` can be used to find the beginning of memory for an object.

The discussion of smart pointers in Item 28 is based in part on Steven Buroff's and Rob Murray's *C++ Oracle* column in the October 1993 *C++ Report*; on Daniel R. Edelson's classic paper, "Smart Pointers: They're Smart, but They're Not Pointers," in the proceedings of the 1992

USENIX C++ Conference; on section 15.9.1 of Bjarne Stroustrup's *The Design and Evolution of C++* (see page 285); on Gregory Colvin's "C++ Memory Management" class notes from C/C++ Solutions '95; and on Cay Horstmann's column in the March-April 1993 issue of the *C++ Report*. I developed some of the material myself, though. Really.

In Item 29, the use of a base class to store reference counts and of smart pointers to manipulate those counts is based on Rob Murray's discussions of the same topics in sections 6.3.2 and 7.4.2, respectively, of his *C++ Strategies and Tactics* (see page 286). The design for adding reference counting to existing classes follows that presented by Cay Horstmann in his March-April 1993 column in the *C++ Report*.

In Item 30, my discussion of lvalue contexts is based on comments in Dan Saks' column in the *C User's Journal* (now the *C/C++ Users Journal*) of January 1993. The observation that non-proxy member functions are unavailable when called through proxies comes from an unpublished paper by Cay Horstmann.

The use of runtime type information to build vtbl-like arrays of function pointers (in Item 31) is based on ideas put forward by Bjarne Stroustrup in postings to `comp.lang.c++` and in section 13.8.1 of his *The Design and Evolution of C++* (see page 285).

The material in Item 33 is based on several of my *C++ Report* columns in 1994 and 1995. Those columns, in turn, included comments I received from Klaus Kreft about how to use `dynamic_cast` to implement a virtual `operator=` that detects arguments of the wrong type.

Much of the material in Item 34 was motivated by Steve Clamage's article, "Linking C++ with other languages," in the May 1992 *C++ Report*. In that same Item, my treatment of the problems caused by functions like `strdup` was motivated by an anonymous reviewer.

The Book

Reviewing draft copies of a book is hard — and vitally important — work. I am grateful that so many people were willing to invest their time and energy on my behalf. I am especially grateful to Jill Huchital, Tim Johnson, Brian Kernighan, Eric Nagler, and Chris Van Wyk, as they read the book (or large portions of it) more than once. In addition to these gluttons for punishment, complete drafts of the manuscript were read by Katrina Avery, Don Box, Steve Burkett, Tom Cargill, Tony Davis, Carolyn Duby, Bruce Eckel, Read Fleming, Cay Horstmann, James Kanze, Russ Paielli, Steve Rosenthal, Robin Rowe, Dan Saks, Chris Sells, Webb Stacy, Dave Swift, Steve Vinoski, and Fred Wild. Partial drafts were reviewed by Bob Beauchaine, Gerd Hoeren,

Jeff Jackson, and Nancy L. Urbano. Each of these reviewers made comments that greatly improved the accuracy, utility, and presentation of the material you find here.

Once the book came out, I received corrections and suggestions from many people: Luis Kida, John Potter, Tim Uttormark, Mike Fulkerson, Dan Saks, Wolfgang Glunz, Clovis Tondo, Michael Loftus, Liz Hanks, Wil Evers, Stefan Kuhlins, Jim McCracken, Alan Duchan, John Jacobsma, Ramesh Nagabushnam, Ed Willink, Kirk Swenson, Jack Reeves, Doug Schmidt, Tim Buchowski, Paul Chisholm, Andrew Klein, Eric Nagler, Jeffrey Smith, Sam Bent, Oleg Shteynbuk, Anton Doblmaier, Ulf Michaelis, Sekhar Muddana, Michael Baker, Yechiel Kimchi, David Papurt, Ian Haggard, Robert Schwartz, David Halpin, Graham Mark, David Barrett, Damian Kanarek, Ron Coutts, Lance Whitesel, Jon Lachelt, Cheryl Ferguson, Munir Mahmood, Klaus-Georg Adams, David Goh, Chris Morley, Rainer Baumschlager, Christopher Tavares, Brian Kernighan, Charles Green, Mark Rodgers, Bobby Schmidt, Sivaramakrishnan J., Eric Anderson, Phil Brabbin, Feliks Kluzniak, Evan McLean, Kurt Miller, Niels Dekker, Balog Pal, Dean Stanton, William Mattison, Chulsu Park, Pankaj Datta, John Newell, Ani Taggu, Christopher Creutzi, Chris Wineinger, Alexander Bogdanchikov, Michael Tegtmeyer, Aharon Robbins, Davide Gennaro, Adrian Spermezan, Matthias Hofmann, Chang Chen, John Wismar, Mark Symonds, Thomas Kim, and Ita Ryan. Their suggestions allowed me to improve *More Effective C++* in updated printings (such as this one), and I greatly appreciate their help.

During preparation of this book, I faced many questions about the emerging ISO/ANSI standard for C++, and I am grateful to Steve Clamage and Dan Saks for taking the time to respond to my incessant email queries.

John Max Skaller and Steve Rumsby conspired to get me the HTML for the draft ANSI C++ standard before it was widely available. Vivian Neou pointed me to the Netscape WWW browser as a stand-alone HTML viewer under (16 bit) Microsoft Windows, and I am deeply grateful to the folks at Netscape Communications for making their fine viewer freely available on such a pathetic excuse for an operating system.

Bryan Hobbs and Hachemi Zenad generously arranged to get me a copy of the internal engineering version of the MetaWare C++ compiler so I could check the code in this book using the latest features of the language. Cay Horstmann helped me get the compiler up and running in the very foreign world of DOS and DOS extenders. Borland (now Inprise) provided a beta copy of their most advanced compiler, and Eric Nagler and Chris Sells provided invaluable help in testing code for me on compilers to which I had no access.

Without the staff at the Corporate and Professional Publishing Division of Addison-Wesley, there would be no book, and I am indebted to Kim Dawley, Lana Langlois, Simone Payment, Marty Rabinowitz, Pradeepa Siva, John Wait, and the rest of the staff for their encouragement, patience, and help with the production of this work.

Chris Guzikowski helped draft the back cover copy for this book, and Tim Johnson stole time from his research on low-temperature physics to critique later versions of that text.

Tom Cargill graciously agreed to make his *C++ Report* article on exceptions (see page 287) available at the Addison-Wesley Internet site.

The People

Kathy Reed was responsible for my introduction to programming; surely she didn't deserve to have to put up with a kid like me. Donald French had faith in my ability to develop and present C++ teaching materials when I had no track record. He also introduced me to John Wait, my editor at Addison-Wesley, an act for which I will always be grateful. The triumvirate at Beaver Ridge — Jayni Besaw, Lorri Fields, and Beth McKee — provided untold entertainment on my breaks as I worked on the book.

My wife, Nancy L. Urbano, put up with me and put up with me and put up with me as I worked on the book, continued to work on the book, and kept working on the book. How many times did she hear me say we'd do something after the book was done? Now the book is done, and we will do those things. She amazes me. I love her.

Finally, I must acknowledge our puppy, Persephone, whose existence changed our world forever. Without her, this book would have been finished both sooner and with less sleep deprivation, but also with substantially less comic relief.

引 言

对C++程序员而言，现在是令人振奋的时代。尽管C++商业化尚不足10 年，却已然成为几乎所有主要计算平台的系统编程语言。越来越多的面临挑战性编程问题的公司和个人不断投入C++的怀抱，那些尚未使用C++的人们则通常被问及何时（而非是否）开始使用C++。C++标准化工作本质上已经完成，其附带的标准库范围之广（涵盖并胜过C标准库），使我们得以在不牺牲移植性或不必从头实现常用算法和数据结构的情况下编写出丰富的复杂程序。C++编译器数量不断增加，它们提供的语言特性持续扩张，产生的代码质量也不断得到改善。用于C++程序开发的工具和环境越发丰富、强大且健壮。商业库几乎可以消除在很多应用领域中编写代码的需要。

由于C++语言已经成熟并且我们对其使用经验日益增多，我们需要的信息也发生了变化。在1990 年，人们希望知道C++是什么。而到了1992 年，他们则希望知道如何使用它。今天，C++程序员则提出了更高级的问题：如何设计软件才能使其适应将来的需要？如何在不危及正确性和易用性的前提下提高代码的效率？如何实现语言未直接提供支持的复杂功能？

在本书中，我将回答这些问题以及其他许多诸如此类的问题。

本书向你展示如何设计和实现更有效的C++软件，即，行为的正确性有着更好的保证、发生异常时表现更为健壮、更高效、移植性更好、更好地运用了语言特性、更优雅地适应变化、在混合语言环境中工作得更好、更易被正确使用、更不易被误用的C++软件。简而言之，就是设计和实现出更好的软件。

本书内容被划分为35个条款。每一个条款都总结了C++编程社群在特定主题上的智慧积累。大部分条款以指导方针的形式呈现，伴随每一个方针的解说则描述了该方针为何存在、倘若不遵循该方针将会发生什么后果，以及在什么情形下你有理由违反该方针。

这些条款被分为几大类。一些条款关注特定的语言特性，尤其是你可能缺乏使用经验的较新特性。例如，条款9~条款15专注于异常主题。另外一些条款则解释如何结合运用语言的各色特性以实现更高级的目标。例如，条款25~条款31描述如何约束对象创建的个数和地点，如何根据一个以上的对象类型创建表现出“虚拟性”的函数，如何创建“智能指针”，等等。还有其他一些条款讨论更广泛的主题，条款16~条款24专注于讨论效率。不论一个特定的条款讨论的是什么主题，它们都提供了具有实效的途径。在本书中，你将学习到如何更有效地使用C++。那些构成大多数C++教材的语言特性描述，在本书中只是作为背景信息出现。

这种讲解方式意味着在阅读本书之前你就应该熟悉C++。这里假定你已了解类、保护级别、虚函数和非虚函数等，还假定你已经熟悉模板和异常背后蕴藏的理念。但我并不期望你是一位语言专家，所以当触及不那么为人熟知的C++特性时，我总会给出必要的解释。

本书所述的C++

本书中描述的C++是1998年国际标准委员会定义的C++语言。这意味着我可能使用了你手头的编译器尚不支持的一些语言特性。别担心，我猜对你而言唯一的“新”特性应该是模板，但现在几乎所有编译器都提供了对它的支持。我还使用了异常，但主要局限于条款9~条款15，这几个条款特别专注于讨论异常。如果你手头的编译器不支持异常机制，没关系，这并不会影响你学习本书其余部分内容。进一步而言，即便你手头没有支持异常的编译器，也应该阅读条款9~条款15，因为这些条款检视了任何情况下你都需要理解的议题。

我承认，仅凭标准委员会授意某一语言特性或认可某种实践，并不能保证该语言特性已得到目前编译器的支持，或该实践可应用于已有的开发环境中。当面临理论和实践之间的差异时，我对两者都加以讨论，尽管我更偏向于可以工作的实践。正因为两者都进行讨论，所以当你的编译器和C++标准不一致时，本书可以助你一臂之力，并向你展示如何使用现有构造来模拟你手头的编译器尚未支持的语言特性。当你决定将一些迂回方式转换为新支持的语言特性时，本书亦将给你指导。

由于不同的编译器实现对C++标准的遵从度不同，因此建议你至少在两种编译器环境下编写代码。这有助于让你避免无意中依赖于某个厂商的专有语言扩展或它对标准的曲解，还有助于让你避免使用“新锐”编译器技术（例如，只有一家厂商提供的新语言特性支持）。此类语言特性通常实现得不够好（充满bug或速度慢，或兼而有之），而且在对它们的介绍方面，C++社群尚缺乏经验，无法为你提供如何正确地使用它们的忠告。摧枯拉朽固然令人兴奋，但当你的目标是生产可靠的代码时，最好还是能够让他人在一头扎入之前先试试水之深浅。

你将在本书中看到两个你可能不太熟悉的C++构造，两者都是相对较晚出现的语言扩展。一些编译器支持它们，但如果你的编译器不支持，可以很容易地利用你熟悉的特性模拟之。

第一个构造是bool类型，其值为关键字true或false。如果你的编译器尚未实现bool，有两种模拟方式。其一是使用全局枚举：

```
enum bool { false, true };
```

这种方式允许你根据函数带有一个bool还是int对其进行重载，缺点是内建的比较操作符（即==、<、>=等）仍然传回int。结果导致如下代码的行为不像我们预期的那样：

```
void f(int);
void f(bool);
int x, y;
...
f( x < y );   // 调用f(int)，其实应该调用f(bool)
```

当你将代码提交给真正支持bool类型的编译器时，这种采用枚举模拟bool的方式可能会导致代码的行为发生改变。

另一种替代方式是使用typedef来定义bool，并以常量对象表示true和false：

```
typedef int bool;
const bool false = 0;
const bool true = 1;
```

这种方式和传统的C/C++语义兼容，并且当使用这种模拟方式的程序被移植到一个支持bool类型的编译器时，其行为不会发生改变。缺点是，当对函数进行重载时无法区分bool和int。总之，这两种模拟方法都有道理，请选择最适合你所处环境的那一种。

第二个新构造其实包含四个构造，即转型操作符static_cast、const_cast、dynamic_cast以及reinterpret_cast。如果你不熟悉这些转型操作，请翻到条款2并阅读全部内容。它们不仅仅比所取代的C风格的转型做得更多，而且做得更好。在这本书中，任何时候当需要执行一个转型操作时，我都使用这些新风格的转型操作符。

C++拥有的不仅仅是语言自身，它还有一个标准库。只要有可能，我就会使用标准string类来代替原生的char*指针，并且建议你也这么做。string对象并不比基于char*的字符串难用，而且还使你避免考虑大部分内存管理问题。此外，如果抛出了异常，string对象不大会发生内存泄漏问题（见条款9 和条款10）。一个有着良好实现的string类的效率可以与其char*等价物的效率相媲美，甚至更好（参见条款29，洞察是如何做到这一点的）。如果你手头无标准string类可用，当然可以使用类似于string的类。建议使用它，因为几乎任何东西都比原生的char*要好。

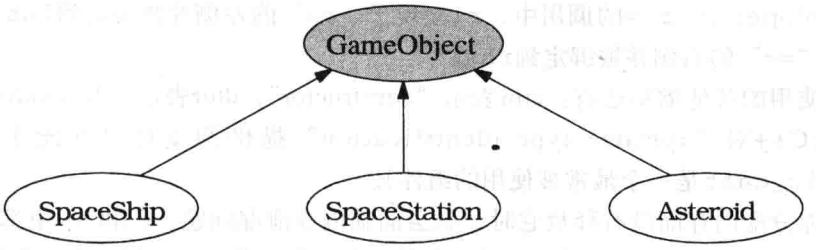
任何时候只要有可能我就会使用来自标准库的数据结构。这类数据结构取自标准模板库（Standard Template Library，即STL，见条款35）。STL包含bitsets、vectors、lists、queues、stacks、maps、sets以及更多的东西，应该优先使用这些标准化的数据结构，而不要抵制不住诱惑自己编写一个特殊的版本。你的编译器也许没有携带STL，但不要因此就不使用它。感谢SGI，你可以从SGI的STL Web站点（http://www.sgi.com/tech/stl/）下载一份免费的拷贝，它可以和许多编译器协作。

如果你目前正使用一个包含各种算法和数据结构的库并且感到满意，那就不要仅仅因为STL“标准”就转而使用它。然而，如果你在使用一个STL组件和从头编写

你自己的代码之间进行选择，那么应该选择使用STL。还记得代码复用吗？STL（以及标准库的其他组件）中有大量很值得复用的代码。

约定与术语

在本书中，任何时候当我提到继承时，都是指公有继承。如果不是指公有继承，我会明确地予以说明。当绘制继承层次结构图时，我通过绘制从派生类指向基类的箭头来描述基类和派生类之间的关系。例如，以下是一幅来自条款31的继承层次结构图：



这种表示法和我在《Effective C++》第1版（不是第2版）中使用的约定相反，但我现在可以确信大多数C++实践者都是绘制从派生类指向基类的继承箭头线，我很高兴随大流。在这类图形中，抽象类（例如GameObject）被加上阴影而具体类（例如SpaceShip）则未加阴影。

继承导致对象的指针和引用具有两种不同的类型，分别是静态类型和动态类型。指针或引用的静态类型是指其声明时的类型。动态类型则由它实际指向的对象的类型确定。下面是基于上述类层次结构编写的一些例子代码：

```
GameObject *pgo =                  // pgo的静态类型是GameObject*，
  new SpaceShip;                   // 动态类型是SpaceShip*
Asteroid *pa = new Asteroid;       // pa的静态类型是Asteroid*，
                                   // 动态类型也是Asteroid*
pgo = pa;                          // pgo的静态类型仍然是（并且总是）GameObject*，
                                   // 动态类型现在变成了Asteroid*
GameObject& rgo = *pa;             // rgo的静态类型是GameObject，
                                   // 动态类型是Asteroid
```

这些例子也示范了我喜欢的一种命名约定。pgo是一个指向GameObject的指针，pa是一个指向Asteroid的指针，rgo是一个指向GameObject的引用。我通常以这种方式为对象的指针和引用命名。

我特别喜欢的两个参数名称是lhs和rhs，它们分别是“left-hand side”和“right-hand side”的缩写。为了理解这些名字背后的理念，考虑一个用于表示有理数

的类：

```
class Rational { ... };
```

如果我想要一个用于比较一对Rational对象的函数，可以将其声明如下：

```
bool operator==(const Rational& lhs, const Rational& rhs);
```

这让我能够写出如下所示的代码：

```
Rational r1, r2;
...
if (r1 == r2) ...
```

在对operator==的调用中，r1出现于“==”的左侧并被绑定到lhs，而r2则出现在“==”的右侧并被绑定到rhs。

我使用的其他缩写还有：ctor表示“constructor”，dtor表示“destructor”，RTTI则表示C++对“runtime type identification”提供的支持（在此上下文中，dynamic_cast是一个最常被使用的组件）。

当你分配内存而没有释放它时，就会面临内存泄漏问题。C和C++中都存在内存泄漏问题，但在C++中内存泄漏“泄漏”的可能不仅仅是内存。因为当对象被创建时，C++会自动调用其构造函数，而构造函数自身可能会分配资源。例如，考虑以下代码：

```
class Widget { ... };          // 某个类（它是什么并不重要）
Widget *pw = new Widget;       // 动态分配一个Widget对象
...                            // 假设pw从未被删除
```

这段代码会泄漏内存，因为pw指向的Widget对象从未被删除。而且，如果Widget构造函数分配了“本应在Widget对象被销毁时释放”的附加资源（例如，文件描述符、信号量、窗口句柄以及数据库锁等），那么这些资源也如同内存那样丢失了。为了强调在C++中内存泄漏往往也会泄漏其他资源，在本书中，我通常说资源泄漏而不是内存泄漏。

你将不会在这本书中看到很多内联函数。这并不是因为我不喜欢内联。恰恰相反，我坚信内联函数是C++的一个重要特性。然而，由于用于决定一个函数是否应被内联的准则可能会很复杂、微妙且与平台有关，因此我尽量避免内联，除非有一个我期望进行内联的关键之处。当你在本书中看到一个非内联函数时，并不意味着我认为把它声明为inline是个坏主意，只是因为是否内联该函数与我正在讨论的主题无关。

有一些C++特性已经被标准委员会摒弃。这样的特性将最终被从语言中移除，因为较新的特性已经加入，它们可以做被废弃的特性所做的工作，而且做得更好。在这本书中，我将会介绍被摒弃的构造，并说明取代它们的语言特性。应该尽量避免使用被废弃的特性，但也没理由过度在意对它们的使用。原因在于，为了为顾客保持向后兼容性，编译器厂商往往会支持废弃的特性很多年。

书中所言的客户（Client）是指使用你编写的代码的人（程序员）或物（通常指类或函数）。例如，如果你编写了一个Date类（用于表示生日、最后期限等），任何使用该类的人就是你的客户，此外，任何使用了Date类的代码片断也是你的客户。客户很重要。实际上，客户正是问题实质之所在！道理很简单，如果没有人使用你编写的软件，那又编写它作甚？你会发现我处心积虑让客户的日子好过一些，通常这会让你的日子更难过，因为优秀的软件总是以客户为中心，客户就是上帝。如果这种说法让你感觉我用情太滥，不妨从利己主义的角度考虑一下。你曾使用过自己编写的类或函数吗？如果是，你就是你自己的客户。所以让客户更轻松，通常也就是让自己更轻松。

当讨论类模板或函数模板以及由它们产生的类或函数时，我保留了偷懒的权利，对模板及其实例之间的差别不加区分。例如，如果Array是一个接受类型参数T的类模板，我可能将该模板的特定实例称为Array，尽管Array<T>才是该类的真正的名字。类似地，如果swap是一个接受类型参数T的函数模板，我可能以swap取代swap<T>来表示其实例。为了防止在某些情况下这种速记法不够清晰，我会在谈到模板实例时带上模板参数。

报告bug、提供建议、获取更新

我已尽力使这本书精确、可读性好、有用，但我知道它必定有改善的余地。如果你发现任何种类的错误，不管是技术性的、语言上的、排版方面的，抑或任何其他方面的，请告诉我。我将努力在本书重印时予以纠正。如果你是某个错误的第一个报告者，我将很高兴将你的大名加入本书的致谢辞中。如果你有其他改善建议，我同样欢迎。

我将继续收集在C++中有效地编程的指导方针。如果你有新指导方针的想法并愿意与我分享，我将非常高兴。请将你的指导方针、评论、批评以及bug报告邮寄到以下地址：

Scott Meyers
c/o Editor-in-Chief, Corporate and Professional Publishing
Addison-Wesley Publishing Company
1 Jacob Way
Reading, MA 01867
U. S. A.

或者，你也可以发送电子邮件到mec++@awl.com。

我维护着一份自本书首次印刷以来的修订列表，其中包括错误修正、文字澄清以及技术更新。这个列表，连同其他相关信息，可从本书网站（http://www.awl.

com/cp/mec++.html）获得。你也可以通过匿名FTP从ftp.awl.com 的cp/mec++目录中获取。如果你希望拥有这份修订列表，但无法上网，请向以上地址发信申请，我会寄一份给你。

如果你希望当我对本书做出修改时得到通知，可以考虑加入我的邮件列表，请访问http://www.aristeia.com/MailingList/index_frames.html。

闲话少说，让我们开始揭示之旅！