TURING 图灵程序设计丛书

Addison Wesley

Exceptional C++ Style

40 New Engineering Puzzles, Programming Problems, and Solutions

C++编程剖析

问题、方案和设计准则

[美] Herb Sutter 著

刘未鹏 译

- C++标准委员会主席力作
- 最负盛名的Exceptional系列之一
- 幽默、辛辣、揭示C++的长处与缺陷

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Exceptional C++ Style

40 New Engineering Puzzles, Programming Problems, and Solutions

C++编程剖析

问题、方案和设计准则

[美] Herb Sutter 著

刘未鹏 译

人民邮电出版社

北京

图书在版编目（CIP）数据

C++编程剖析 ：问题、方案和设计准则 / （美）萨特(Sutter,H.）著 ；刘未鹏译. -- 2版. -- 北京 ：人民邮电出版社，2011.1
（图灵程序设计丛书）
书名原文：Exceptional C++ Style:40 New Engineering Puzzles,Programming Problems,and Solutions
ISBN 978-7-115-24099-6

Ⅰ. ①C… Ⅱ. ①萨… ②刘… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第215267号

内 容 提 要

本书中，C++大师Herb Sutter通过40个编程问题，使读者不仅“知其然”，更要“知其所以然”，帮助程序设计人员在软件中寻找恰到好处的折中，即讨论如何在开销与功能之间、优雅与可维护性之间、灵活性与过分灵活之间寻找完美的平衡点。本书是围绕实际问题及其解决方案展开论述的，对一些至关重要的C++细节和相互关系提出了新的见解，为当今关键的C++编程技术（如泛型编程、STL、异常安全等）提供了新的策略。本书的目标是让读者在设计、架构和编码过程中保持良好的风格，从而使编写的C++软件更健壮、更高效。本书适合中高级C++程序员阅读。

图灵程序设计丛书

C++编程剖析
问题、方案和设计准则

◆ 著 [美] Herb Sutter
译 刘未鹏
责任编辑 朱 巍
执行编辑 毛倩倩

◆ 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
中国铁道出版社印刷厂印刷

◆ 开本：800×1000 1/16
印张：18.25
字数：429千字 2011年1月第2版
印数：6 001－9 000册 2011年1月北京第1次印刷
著作权合同登记号 图字：01-2005-3577号
ISBN 978-7-115-24099-6

定价：49.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

译 者 序

逍遥派武功讲究轻灵飘逸，闲雅清隽，丁春秋和虚竹这一交上手，但见一个童颜白发，宛如神仙，一个僧袖飘飘，冷若御风。两人都是一沾即走，当真便似一对花间蝴蝶，蹁跹不定，于这“逍遥”二字发挥到了淋漓尽致。旁观群雄于这逍遥派的武功大都从未见过，一个个看得心旷神怡，均想：“这二人招招凶险，攻向敌人要害，偏生姿势却如此优雅美观，直如舞蹈。这般举重若轻、潇洒如意的掌法，我可从来没见过……”

—— 金庸 《天龙八部》

金庸小说中描绘的逍遥派武功讲究的是飘逸灵动，然则绝非片面追求招式漂亮，招招看起来都优雅美观，但招招都攻往要害。

写代码也应如此。

毫无疑问，代码的目的是实现既定的功能，所以实用论者可能会说代码只需实现既定功能，无需费时费事去搞那么多周折、弄那么多形式。如果代码写完就可以永远不用再管，这番论断倒是言之有理的。然而事实是，代码的维护占用了软件开发中的大部分乃至绝大部分时间和人力。譬如说，做外包业务的公司大部分时间都是在维护代码或者说改代码，而不是原生开发，这便意味着外来的代码形式“漂亮”与否直接关系到一个项目的开销。另外，一般的公司，就算是作坊公司，日积月累，也一定会有自己的代码库、自己的遗留代码，只要他们想节省开销，复用以前写成的代码是必由之路，因此这里代码的可读性、代码结构的可扩展性等就变得异常重要了。有过一些经验的程序员都知道维护别人（甚至自己）的代码是最痛苦的事情。可见代码的“形式”构成了软件质量的一个重要部分。无论如何，就像书上常说的：“内容决定形式，但形式对内容也有反作用。”

然而徒有其表的代码只是个花架子，没有真材实料，再花哨的形式也无济于事。软件归根到底要的还是功能。这就好比练武，光有花拳绣腿而没有扎实的内功是万万成不了气候的。

那么，在软件开发领域，这两者到底矛不矛盾呢？答案是根本不矛盾。我们之所以平时无法鱼和熊掌兼得，一方面固然是由于进度紧的缘故，另一方面也与编码时的方法学有一定关系。

本书讨论的正是后者。

正是由于C++是一门非常自由的语言，因此C++编码的“形式”才变得异常重要，以至于一

些大公司都规定了各自的编码标准。本书正是着眼于C++编码风格的一本书，但这里所谓的编码风格并非指命名风格、注释风格之类浅显的东西，而是指在某些特定的问题领域所采取的编码方式（或“形式”）。在C++中往往有若干条道路都能通往同一个终点，问题是选择哪条道路才最具有远见，最能达到形式和内容的统一，这一点很关键。

为此，本书的40个条款围绕C++日常编码中的种种问题展开讨论，详细考察各解决方案之间的优劣，最终给出权衡之下最为妥帖的方案，并将其提炼为一条条的编码准则。

武侠小说的读者大多数都希望看到招式漂亮而又非常厉害的武功，觉着很过瘾。殊不知维护代码的程序员何尝不希望看到写得漂亮而又有实在功能的代码呢？

相信你在Sutter的书中能够找到一些答案。

最后，感谢老朋友谢轩和罗翼，与他们讨论问题是我的乐事。谢轩（《Symbian OS C++高效编程》译者）提供了第34条的译稿，罗翼则无偿帮我初译了第六部分。他们的热情给了我莫大的帮助，他们的技术和文笔也让我获益颇多☺。

感谢父母一直以来的支持，令我不敢懈怠☺。

前　言

布达佩斯，匈牙利的首都。一个炎热的夏日傍晚。穿过美丽的多瑙河望去，余晖中的东岸景色优美恬静。

在本书封面上色彩柔和的欧洲风光中，哪栋建筑首先映入你的眼帘？几乎可以肯定，是照片左边的国会大厦。这栋巨大的新哥特式建筑以它优美的圆穹、直插天际的尖塔、不计其数的外墙雕塑以及其他华丽装饰一下攫住了你的目光，而它更引人注目之处在于，它与四周建筑在多瑙河畔那些刻板的实用建筑形成了极其鲜明的对照。

为什么会有这么大的差异呢？一方面，国会大厦是在 1902 年竣工的，而其他味同嚼蜡的建筑则大部分都是在第二次世界大战以后建成的。

“啊哈，”你可能会想，“这的确解释了为什么差异如此之大。然而这与本书到底有什么关系呢？”

毫无疑问，风格的表达与你在表达风格时灌注的哲学和思维方式是有很大关系的，这一点不管对于建筑学还是对于软件架构来说都同样适用。我相信你们都见过像封面上国会大厦那样宏伟而华丽的软件，我同样相信你们也见过仅能工作而且一团乱麻似的软件。极端一点说，我相信你也见过许多过分追求风格反而弄巧成拙的华而不实之作，以及许多只顾尽快完成任务而毫无风格的“丑小鸭”（而且永远也不会变成天鹅）。

风格还是实用

哪个更好？

不要太相信自己知道答案。一方面，除非你给出一个明确的标准，否则“更好”只是一个无意义的评价。对什么更好呢？在哪些方面更好呢？另一方面，答案几乎总是这两者的平衡，最开始总是“这取决于……”。

本书讨论的是如何在使用 C++进行软件设计和实现的诸多细节方面找到最佳平衡点，如何更好地理解所拥有的工具和设施，弄清它们应该在什么时候应用。

快速回答：与四周索然无味的建筑相比，封面上的国会大厦是更好的建筑吗？其建筑风格更好吗？如果不加思索就给出答案，很可能你会说“当然”，但是别忘了，你还没有考虑其建造和修缮的代价呢。

- 建造。在1902年竣工之时，它是当时世界上最大的国会大厦。人们花费了难以想象的时间、不计其数的人力物力来兴建它，以至于许多人称它为“白象”（white elephant），意思是耗资过大的美丽事物。考虑这样一个问题：比较起来，花费同样的投资能够建造多少幢周围那种不美观、单调或许干脆是令人厌烦的实用建筑？如果你是在一个工程进度压力远比这座国会大厦建造时代要大得多的行业工作，你又会怎么做？
- 修缮。你们中那些熟悉这座建筑的人会注意到照片中的建筑正在进行修缮翻新，而且这个工作已经持续了好多年，其间又极有争议地花费了巨额的资金。然而除了最近的这轮昂贵的修缮之外，之前还有多次修缮，因为这座建筑外墙上的精美雕刻所用的材料并不合适，太过柔软了。在大厦建成后不久，这些雕刻就必须不断修缮，它们逐渐被替换为更为坚固而耐久的材料。这些华丽之物的大规模修缮自从20世纪初开始就一直没停过，持续了近一个世纪。

软件开发中的情形也与此类似，重要的是在建造的代价和获得的功能之间、在优雅与可维护性之间、在发展的潜在可能与过分追求华丽之间寻求合理的平衡。

使用C++来进行软件设计和架构时，我们每天都得面对这些类似的权衡。在本书讨论的问题当中有这样几个问题：使代码成为异常安全的就意味着将它变得更好了吗？如果是这样的，那么这里所谓的“更好”是指什么意义上的呢？什么时候它可能不是“更好”的呢？在本书中你会得到明确的答案。封装呢？封装是否令软件变得更好？为什么？什么时候封装反倒不能令软件变得更好？如果你想知道答案，继续往下读。内联是一项有益的优化吗？内联是什么时候进行的呢？（你在回答这个问题的时候可得十分小心了。）C++中的模板导出（export）特性与封面上的国会大厦有什么相通之处呢？std::string与多瑙河畔的巨型建筑又有何相通之处呢？

最后，在考虑了许多C++技术和特性之后，我们会用最后一部分来考察摘自公开发布的著名代码中的几个实际例子，看看代码的作者在哪些方面做得好，在哪些方面做得不好，以及什么样的替代方案可能在实用性与良好的C++风格之间取得更好的平衡。

我希望本书以及 *Exceptional C++*系列的其他图书能够开阔你的视野，增加你有关许多细节及其相互关系的知识，让你进一步了解到如何在编写自己的软件时找到合理的平衡点。

请再看一眼封面上的照片，在照片的右上方，你会看到一个热气球。如果我们乘坐那样的热气球飞越城市的上空，整个城市的景色将尽收眼底。我们会看到风格跟实用是如何相互影响、相互依存的，我们也会知道如何去进行权衡并找到合理的平衡点，所有的决策将各得其所，构成一个富于生机的整体。

是的，我想布达佩斯是一个伟大的城市——充满着丰富的历史底蕴，充满着不尽的神秘喻义。

伟大的苏格拉底

古希腊哲学家苏格拉底通过提问来达到教学目的。他精心准备的问题是为了引导并帮助学生从已知的知识引出结论，并说明他们所学的东西是如何彼此相关、如何与他们现有的知识有着千

丝万缕的联系的。这种教学方式后来广为人知，我们称它为“苏格拉底方法”。苏格拉底的这种著名的教学方法能够吸引学生，让学生思考，并帮助学生从已知出发去引出新的东西。

本书与它的前面几本书（*Exceptional C++* [Sutter00]和 *More Exceptional C++* [Sutter02]）一样，正是借鉴了苏格拉底的做法。本书假定你在编写 C++产品代码方面已有一些经验，书中使用了一种问答的形式来告诉你如何有效地利用标准 C++及其标准库，特别地，我们将关注的中心放在如何用现代 C++开发可靠的软件上。书中的许多问题都是从我以及其他人在编写 C++产品代码时遇到的问题当中提炼出来的。问题的目标是帮助你从已知的以及刚学到的东西出发得出结论，并展示它们之间如何关联。书中给出的问题会展示如何对C++设计和编程问题做出理性的分析和判断，其中有些只是常见问题，有些不是那么常见；有些是非常单纯的问题，而有些则深奥一些；另外还有几个问题之所以放在书中只是因为……因为它们比较有趣。

本书涉及了 C++的方方面面。我的意思并不是说它触及了 C++的每个细枝末节（那可需要多得多的篇幅了），我只不过是说它是从 C++语言和库特性这块大调色板上取色，并描绘出一幅图景，展示那些看似毫无瓜葛的特性如何编织到一起，从而构成常见问题的一个个漂亮解决方案。另外，本书还展示了那些看似无关的部分是如何互相之间盘根错节、存在着千丝万缕的联系的（即便有时你也许并不希望它们之间有什么联系），以及如何处理这些复杂关系。你会看到一些关于模板和名字空间的讨论，也会看到一些关于异常与继承的讨论，同样，另外还有关于坚实的类设计和设计模式的讨论，关于泛型编程与宏技巧的讨论，等等。此外，还有一些实实在在的（而不是一些花边新闻式的边栏小字）条款，展示现代 C++中所有这些部分之间的相互关系。

本书遵循了 *Exceptional C++*和 *More Exceptional C++*两本书的传统：它通过短小精悍的条款的组织形式，并将这些条款再进一步分组为一个个的主题来介绍新知识。读过我的第一本书的读者会发现一些熟悉的主题，不过现在包含了新的东西，诸如异常安全、泛型编程以及优化和内存管理技术。我的几本书在主题上有部分重叠，但内容上并没有重复。本书沿袭了对泛型编程和高效使用标准库的一贯强调态度，包括一些重要的模板和泛型编程技术的讨论。

书中的大多数条款最初出现在杂志专栏上以及网上，尤其是我为 *C/C++ Users Journal* 和 *Dr. Dobb's Journal*、已停刊的 *C++ Report* 以及其他期刊所写的专栏文章，另外还有我的 *Guru of the Week*[GotW]问题 63 到 86。不同的是本书中的材料与最初的版本相比经过了重大的修订、扩展、校正和更新，因此这本书（以及 www.gotw.ca 网站上的勘误表）应该被当成原先那些文章的最新而且权威的版本。

预备知识

我假定读者已经知道一些 C++的基础知识。如果不是这样，那就先去阅读一些好的关于 C++介绍和概述的文章或书籍。像 Bjarne Stroustrup 的 *The C++ Programming Language* [Stroustrup00]或者 Stan Lippman 和 Josée Lajoie 合著的 *C++ Primer*（第 3 版）[Lippman98][①]，这样的经典是非常

① 此书第 4 版中文版已由人民邮电出版社出版。——编者注

不错的选择。接下来，一定要选择一本 Scott Meyers 的经典书籍(*More*)*Effective C++*[Meyers96, Meyers97]这样的风格指南，我发现这两本书基于 Web 浏览方式的 CD 版本[Meyers99]比较方便实用。

如何阅读本书

本书中的每一条都是以一个谜题或问题来展开的，都有一个介绍性的标题，如下所示。

第##条

难度系数：#

一段简短的介绍性文字，说明该条将要讨论的内容。

标题大致告诉你本条讨论的是什么，通常后面会跟有介绍性的或回顾性的问题（初级问题，原文 JG 是指新来的、级别较低的少尉军官），然后就是主要问题（即专家级问题）。注意，难度系数只是我针对特定主题对大多数读者而言的难度所做的一个大致推测，这就是说你可能会发现一个难度系数为 7 的问题对你来说却比一个难度系数为 5 的问题要来得简单。实际上我的前两本书：*Exceptional C++* [Sutter00]和 *More Exceptional C++* [Sutter02]就曾不断地收到一些读者来信说："嗨！第 *N* 条比它实际上要更难（简单）!" 不同的人对于"简单"的评价标准各有不同。所谓难度系数只是因人而异的，任何条款的难度实际上都取决于你所掌握的知识和经验，而其他人则可能觉得它更容易或更难。不过大多数情况下应当将我给出的难度系数作为一个合理的指示，让你能够知道下面会出现什么问题。

你可能会选择从头至尾按顺序阅读本书，这很好，但是不是非要这么做。你可能决定读某个特定部分的所有条款，因为对该部分的主题特别感兴趣，这也没关系。一般来说书中的所有条款都是基本独立的，除非标注有"之一"、"之二"等的条款之间才会有紧密联系。因此在阅读本书的时候完全可以以跳跃式的方式，顺着条款中的交叉引用（包括对我前两本书的引用）来阅读。唯一需要注意的地方就是，标注了"之几"的连续几个章节之间互有关联，构成了一个主题，除此之外其他条款你完全可以自由选择阅读顺序。

除非我注明某段代码是一个完整的程序，否则它就不是。记住，代码示例通常只是从一个完整程序中摘取出来的一小段代码，不要期望它们都能够独立编译。一般来说你得为其添上一副骨架才能够使其成为一个完整的可编译的程序。

最后，关于书中的 URL。网上的东西总是在变，尤其是那些我无权干涉的东西。因此随意将某些网站地址放在一本刊印的书籍中可不大妥当，因为恐怕在书付印之前有些地址就已经作废了，更不用说这本书在你书架上躺了几年之后了。所以说，当我在书中引用其他人的文章或网址的时候，我给出的地址都链接到我自己的网站（即 www.gotw.ca）上的相关内容，这是我可以控制的，因此我可以在我网站上的相关网页上随时做相应更新，让其中的相关地址指向实际存在的网页。几乎所有在书中引用到的其他人的作品我都放在参考书目里了，而且在我的网站上也放置

了一份副本，其中的链接都是有效的。如果你发现本书中的链接无效了，请发电子邮件向我告知，我会在网站上更新相关的链接（如果我可以找到新地址的话），或者注明链接已经失效（如果我无法找到新地址的话）。无论如何，虽说书一旦印刷便白纸黑字，不可再改，但我网站上的相关内容会保持更新。

致谢

首先我最要感谢的是我的妻子 Tina，感谢她对我一直以来的爱和支持，另外还有我的家庭一直都与我同行，无论我做什么事情。即便是在我熬夜写另一些文章或修改另一些条款的时候，他们也从来都是全力支持的。如果没有他们的耐心和关心，这本书就绝不可能达到现在这个样子。

当然我们的小狗 Frankie 也有一份功劳，她总是在我工作的时候时不时打断我，让我得到宝贵的休息时间。Frankie 对软件架构或语言设计乃至代码微观优化一无所知，但她仍然生活得无比快乐。

我还要感谢丛书编辑 Bjarne Stroustrup，编辑 Peter Gordon、Debbie Lafferty、Tyrrell Albaugh、Bernard Gaffney、Curt Johnson、Chanda Leary-Coutu、Charles Leddy、Malinda McCain、Chuti Prasertsith 以及其他的 Addison-Wesley 团队的编辑们，感谢他们在我写作本书的过程中提供的帮助和坚持。他们是我见过的最好的团队，他们的热情和合作精神使我对这本书的所有设想都成为了现实，我很开心能和他们合作。

另外我还要感谢专家评审小组，他们总是一针见血毫不留情地指出书中的纰漏并给出洞见。他们的努力让你手中的这本书更完整、更可读，也更有用，而光靠我一个人的能力是远远做不到这一点的。尤其要感谢他们向丛书编辑 Bjarne Stroustrup 提供的技术反馈。此外还要感谢 Dave Abrahams、Steve Adamczyk、Andrei Alexandrescu、Chuck Allison、Matt Austern、Joerg Barfurth、Pete Becker、Brandon Bray、Steve Dewhurst、Jonathan Caves、Peter Dimov、Javier Estrada、Attila Fehér、Marco Dalla Gasperina、Doug Gregor、Mark Hall、Kevlin Henney、Howard Hinnant、Cay Horstmann、Jim Hyslop、Mark E. Kaminsky、Dennis Mancl、Brian McNamara、Scott Meyers、Jeff Peil、John Potter、P. J. Plauger、Martin Sebor、James Slaughter、Nikolai Smirnov、John Spicer、Jan Christiaan van Winkel、Daveed Vandevoorde 和 Bill Wade，他们为本书提出了贡献性的见解和建议。当然，书中的错误、遗漏问题和自作聪明的双关语都是我的问题，我负全责。

Herb Sutter

2004 年 5 月于西雅图

目　录

泛型编程与 C++标准库

C++最强大的特性之一就是对泛型编程的支持。C++标准库的高度灵活性就是明证，尤其是标准库中的容器、迭代器以及算法部分（最初也称为 STL）。

与我的另一本书 *More Exceptional C++* [Sutter02]一样，本书的开头几条也是介绍 STL 中一些我们平常熟悉的部件，如 vector 和 string，另外也介绍了一些不那么常见的设施。例如，在使用最基本的容器 vector 时如何避免常见的陷阱？如何在 C++中进行常见的 C 风格字符串操纵？我们能够从 STL 中学到哪些库设计经验（不管是好的、坏的，还是极其糟糕的）？

在考察了 STL 中的模板设施之后，接着讨论关于 C++中的模板和泛型编程的一些更一般性的问题。例如，如何让我们的模板代码避免不必要地（且相当不经意地）损失泛型性。为什么说特化函数模板实际上是个糟糕的主意，而我们又应当怎么替换它？在模板的世界中，我们如何才能正确且可移植地完成像授予友元关系这样看似简单的操作？此外还有围绕着 export 这个有趣的关键字发生的种种故事。

随着我们逐步深入与 C++标准库及泛型编程相关的主题，就会看到关于上述（以及其他）问题的讨论。

第 1 条　vector 的使用

难度系数：4

几乎每个人都会使用std::vector，这是个好现象。不过遗憾的是，许多人都误解了它的语义，结果无意间以奇怪和危险的方式使用它。本条款中阐述的哪些问题会出现在你目前的程序中呢？

初级问题

1. 下面的代码中，注释 A 跟注释 B 所示的两行代码有何区别？

```
void f(vector<int>& v) {
 v[0];        // A
 v.at(0);     // B
}
```

专家级问题

2. 考虑如下的代码：

```
vector<int> v;

v.reserve(2);
assert(v.capacity() == 2);
v[0] = 1;
v[1] = 2;
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
 cout << *i << endl;
}

cout << v[0];
v.reserve(100);
assert(v.capacity() == 100);
cout << v[0];

v[2] = 3;
v[3] = 4;
// ……
v[99] = 100;
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
 cout << *i << endl;
}
```

请从代码的风格和正确性方面对这段代码做出评价。

解决方案

访问vector的元素

1. 下面的代码中，注释 A 跟注释 B 所示的两行代码有何区别？

```
// 示例1-1: [] vs. at
//
void f(vector<int>& v) {
 v[0];        // A
 v.at(0);     // B
}
```

在示例 1-1 中，如果 v 非空，A 行跟 B 行就没有任何区别；如果 v 为空，B 行一定会抛出一个 std::out_of_range 异常，至于 A 行的行为，标准未加任何说明。

有两种途径可以访问 vector 内的元素。其一，使用 vector<T>::at。该成员函数会进行下标越界检查，确保当前 vector 中的确包含了需要的元素。试图在一个目前只包含 10 个元素的 vector 中访问第 100 个元素是毫无意义的，这样做会导致抛出一个 std::out_of_range 异常。

其二，我们也可以使用vector<T>::operator[]，C++98 标准说vector<T>::operator可以、但不一定要进行下标越界检查。实际上，标准对operator[]是否需要进行下标越界检查只字未提，不过标准同样也没有说它是否应该带有异常规格声明。因此，标准库实现方可以自由选择是否为operator[]加上下标越界检查功能。如果使用operator[]访问一个不在vector中的元素，你可就得自己承担后果了，标准对这种情况下会发生什么事情没有做任何担保（尽管你使用的标准库实现的文档可能做了某些保证）——你的程序可能会立即崩溃，对operator[]的调用也许会引发一个异常，甚至也可能看似无恙，不过会偶尔或神秘地出问题。

既然下标越界检查帮助我们避免了许多常见问题，那为什么标准不要求operator[]实施下标越界检查呢？简短的答案是效率。总是强制下标越界检查会增加所有程序的性能开销（虽然不大），即使有些程序根本不会越界访问。有一句名言反映了C++的这一精神：一般说来，不应该为不使用的东西付出代价（或开销）。所以，标准并不强制operator[]进行越界检查。况且我们还有另一个理由要求operator[]具有高效性：设计vector是用来替代内置数组的，因此其效率应该与内置数组一样，内置数组在下标索引时是不进行越界检查的。如果你需要下标越界检查，可以使用at。

调整vector的大小

现在看示例 1-2，该示例对 vector<int>进行了简单操作。

2. 考虑如下的代码：

```
// 示例1-2: vector的一些函数
//
vector<int> v;
```

```
v.reserve(2);
assert(v.capacity() == 2);
```

这里的断言存在两个问题，一个是实质性的，另一个则是风格上的。

首先，实质性问题是，这里的断言可能会失败。为什么？因为上一行代码中对 reserve 的调用将保证 vector 的**容量**至少为 2，然而它也可能大于 2。事实上这种可能性是很大的，因为 vector 的大小必须呈指数速度上升，因而 vector 的典型实现可能会选择总是按指数边界来增大其内部缓冲区，即使是通过 reserve 来申请特定大小的时候。因此，上面代码中的断言条件表达式应该使用>=，而不是==，如下所示：

```
assert(v.capacity() >= 2);
```

其次，风格上的问题是，该断言（即使是改正后的版本）是多余的。为什么？因为标准已经保证了这里所断言的内容，所以再将它明确地写出来只会带来不必要的混乱。这样做毫无意义，除非你怀疑正在使用的标准库实现有问题，如果真有问题，你可就遇到大麻烦了。

```
v[0] = 1;
v[1] = 2;
```

上面这些代码中的问题都是比较明显的，但可能是比较难于发现的明显错误，因为它们很可能会在你所使用的标准库实现上“勉强”能够“正常运行”。

4

大小（size，跟 resize 相对应）跟容量（capacity，与 reserve 相对应）之间有着很大的区别。

- size 告诉你容器中目前实际有多少个元素，而对应地，resize 则会在容器的尾部添加或删除一些元素，来调整容器当中实际的内容，使容器达到指定大小。这两个函数对 list、vector 和 deque 都适用，但对其他容器并不适用。
- capacity 则告诉你最少添加多少个元素才会导致容器重分配内存，而 reserve 在必要的时候总是会使容器的内部缓冲区扩充至一个更大的容量，以确保至少能满足你所指出的空间大小。这两个函数仅对 vector 适用。

本例中我们使用的是 v.reserve(2)，因此我们知道 v.capacity()>=2，这没有问题，但值得注意的是，我们实际上并没有向 v 当中添加任何元素，因而 v 仍然是空的！v.reserve(2)只是确保 v 当中有空间能够放得下两个或更多的元素而已。

准则　记住size/resize以及capacity/reserve之间的区别。

我们只可以使用operator[]()（或at()）去改动那些确实存在于容器中的元素，这就意味着它们是跟容器的大小息息相关的。首先你可能想知道为什么operator[]不能更智能一点，比如当指定地点的元素不存在的时候“聪明地”往那里塞一个元素，但问题是假设我们允许operator[]()以这种方式工作，就可以创建一个有“漏洞”的vector了！例如，考虑如下的代码：

```
vector<int> v;
```