TURING 图灵程序设计丛书

Exceptional C++ Style中文版

40个新的工程难题、编程问题及解决方案

Herb Sutter 著

刘未鹏 译

人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

Exceptional C++ Style 中文版：40 个新的工程难题、编程问题及解决方案 /（美）萨特（Sutter, H.）著；刘未鹏译. —北京：人民邮电出版社，2006.1
（图灵程序设计丛书）

ISBN 7-115-14225-4

Ⅰ. E… Ⅱ. ①萨…②刘… Ⅲ. C 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2005）第 148110 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

Exceptional C++ Style 中文版
40 个新的工程难题、编程问题及解决方案

◆ 著　Herb Sutter
译　刘未鹏
责任编辑　杨海玲

◆ 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
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000　1/16
印张：18.25
字数：408 千字　2006 年 1 月第 1 版
印数：4 001 – 6 000 册　2006 年 4 月北京第 2 次印刷

著作权合同登记号　图字：01-2005-3577 号

ISBN 7-115-14225-4/TP・5107

定价：39.00 元

读者服务热线：(010)88593802　印装质量热线：(010)67129223

版 权 声 明

Authorized translation from the English language edition, entitled *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions,* 1st Edition 0201760428 by Herb Sutter, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2005 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2006.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

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 Explained* 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☺。

刘未鹏
2005年10月
于南京

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二战以后建成的。

“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，和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。人们花费了难以想像的时间、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量的资金。然而除了最近的这轮昂贵的修缮之外，之前还有多次修缮，因为遗憾的是，这座建筑外墙上的精美雕刻是由错误的材料制成的，其材料太过柔软了。在大厦建成后不久，这些雕刻成了人们持续进行修缮的主要内容，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，我们会用一整个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教他的学生们，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。这种教学方式后来变得如此有名，以至于我们称它为“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++设计和编程问题作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）条款用来向你展示现代 C++中所有这些部分之间的相互关系。

Exceptional C++ Style 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]，这样的经典是良

1. 此书第 4 版中文版即将由人民邮电出版社出版。——编者注

好的选择。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：

第##条 **难度：#**

一段简短的介绍性文字，说明该条将要讨论的内容。

条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[JG 问题，JG 是指新来的、级别较低的军官（少尉）]，然后就是主要问题（即 Guru 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作的一个大致推测，这就是说你可能会发现一个难度为 7 的问题对你来说却比一个难度为 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是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。惟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。

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作相应更新，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。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。

当然还有我们的小狗 Frankie 的一份功劳，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。

另外我还要感谢另一群人，就是专家评审小组，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。他们的努力让你现在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，他们在这本书写作的过程中为它的各个部分提出了贡献性的见解和建议。

Herb Sutter

西雅图，2004 年 5 月

目　　录

泛型编程与 C++标准库 …… 1

第 1 条　vector 的使用 …… 2
第 2 条　字符串格式化的“动物庄园”之一：sprintf …… 9
第 3 条　字符串格式化的“动物庄园”之二：标准的（或极度优雅的）替代方案 …… 14
第 4 条　标准库成员函数 …… 23
第 5 条　泛型性的风味之一：基础 …… 26
第 6 条　泛型性的风味之二：够“泛”了吗 …… 30
第 7 条　为什么不特化函数模板 …… 36
第 8 条　友元模板 …… 42
第 9 条　导出限制之一：基础 …… 51
第 10 条　导出限制之二：相互影响，可用性问题以及准则 …… 58

异常安全问题及相关技术 …… 67

第 11 条　try 和 catch …… 68
第 12 条　异常安全性：值得吗 …… 72
第 13 条　对异常规格的实际考虑 …… 75

类的设计、继承和多态 …… 83

第 14 条　顺序，顺序！ …… 84
第 15 条　访问权限的使用 …… 88
第 16 条　（几乎）私有 …… 93
第 17 条　封装 …… 101
第 18 条　虚拟 …… 110
第 19 条　对派生类施加规则 …… 118

内存和资源管理 ····· 129

第 20 条 内存中的容器之一：内存管理的层次 ····· 130
第 21 条 内存中的容器之二：它到底有多大 ····· 133
第 22 条 进行 new 操作，也许会抛出异常之一：new 的方方面面 ····· 140
第 23 条 进行 new 操作，也许会抛出异常之二：内存管理中的实际问题 ····· 148

优化和效率 ····· 155

第 24 条 常量优化 ····· 156
第 25 条 再论内联 ····· 161
第 26 条 数据格式和效率之一：什么时候压缩是真正重要的 ····· 168
第 27 条 数据格式和效率之二：（甚至更少的）位操纵 ····· 172

陷阱、缺陷和谜题 ····· 179

第 28 条 不是关键字的关键字（或者：另一种注释） ····· 180
第 29 条 这是初始化吗 ····· 186
第 30 条 要么 double 要么彻底完蛋 ····· 191
第 31 条 狂乱的代码 ····· 194
第 32 条 小小的拼写错误？鬼画符似的语言以及其他奇形怪状的东西 ····· 199
第 33 条 操作符，无处不在的操作符 ····· 202

风格案例研究 ····· 207

第 34 条 索引表 ····· 208
第 35 条 泛型回调 ····· 218
第 36 条 构造式 union ····· 226
第 37 条 分解 std::string 之一：概观 std::string ····· 242
第 38 条 分解 std::string 之二：重构 std::string ····· 247
第 39 条 分解 std::string 之三：给 std::string 瘦身 ····· 255
第 40 条 分解 std::string 之四：再论 std::string ····· 259

参考文献 ····· 267

索引 ····· 271

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：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？

JG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
}
```

Guru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;
}
```

2 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以确保当前 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。

3

调整vector的大小

现在让我们来看示例 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;
```

```
v.reserve(100);
v[99] = 42; // 错误！但出于讨论的目的，让我们假设这是允许的……

//……这里v[0]至v[98]的值是什么呢？
```

正是因为标准并不强制要求 operator[]()进行区间检查，所以在大多数实现上，v[0]都会简单地返回内部缓冲区中用于存放但尚未存放第一个元素的那块空间的引用。因此 v[0]=1;这行语句很可能被认为是正确的，因为如果接下来输出 v[0](cout<<v[0])的话，或许会发现结果确实是 1，跟（错误的）预期相符合。

再一次提醒，标准并无任何保证说在你使用的标准库实现上一定会出现上述情形，本例只是展示了一种典型的可能情况。标准并没有要求特定的实现在这类情况下（诸如对一个空的 vector v 写 v[0]）该采取什么措施，因为它假定程序员对这类情况有足够的认识。毕竟，如果程序员想要库来帮助进行下标越界检查的话，他们可以使用 v.at(0)，不是吗？

当然，如果将 v.reserve(2)改成 v.resize(2)的话，v[0]=1;v[1]=2;这两行赋值语句就能够顺利工作了。只不过上文中的代码并没有使用 resize()，因此代码并不能保证正常工作。作为一个替代方案，我们可以将这两行语句替换成 v.push_back(1)和 v.push_back(2)，它们的作用是向容器的尾部追加元素，而使用它们总是安全的。 5

```
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
  cout << *i << endl;
}
```

首先，上面这段代码什么都不会打印，因为 vector 现在根本就是空的！这可能会让代码的作者感到意外，因为他们还没意识到其实前面的代码根本就没有往 vector 中添加任何东西。实际上，跟 vector 中的那些已经预留但尚未正式使用的空间“玩游戏”是很危险的。

话虽如此，这个循环本身并没有任何明显的问题，只不过如果在代码审查阶段看到这段代码的话，我会指出其中存在的一些风格上的问题。大多数意见都是初级的，如下所示：

（1）尽量做到 const 正确性。以上的循环当中，迭代器并没有用来修改 vector 中的元素，因此应当改用 const_iterator。

（2）尽量使用!=而不是<来比较两个迭代器。确实，由于 vector<int>::iterator 恰巧是一个随机访问迭代器（当然，并不一定是 int*），因此在这种特定情况下将它跟 v.end()比较是没有任何问题的。但问题是<只对随机访问迭代器有效，而!=对于任何迭代器都是有效的，因此我们应该将使用!=比较迭代器作为日常惯例，除非某些情况下确实需要<（注意，使用!=还有一个好处就是便于将来（如有需要）更改容器类型）。例如，std::list 的迭代器并不支持<，因为它们只不过是双向迭代器。

（3）尽量使用前缀形式的--和++。让自己习惯于写++i 而不是 i++，除非真的需要用到 i 原来的值。例如，如果既要访问 i 所指的元素，又要将 i 向后递增一位的话，后缀形式 v[i++]就比较适用了。

（4）避免无谓的重复求值。本例中 v.end()所返回的值在整个循环的过程中是不会改变的，因此应当避免在每次判断循环条件时都调用一次 v.end()，或许我们应当在循环之前预先将 v.end()求出来。

6

注意，如果你的标准库实现中的 vector<int>::iterator 就是 int*，而且能够将 end()进行内联及合理优化的话，原先的代码也许并无任何额外开销，因为编译器或许能够看出 end()返回的值一直是不变的，从而安全地将求值提到循环外部。这是一种相当常见的情况。然而，如果你的标准库实现的 vector<int>::iterator 并非 int*（例如，在大多数调试版实现当中，其类型都是类类型的），或者 end()之类的函数并没有内联，或者编译器并不能进行相应的优化，那么只有手动将这部分代码提出才能获得一定程度的性能提升。

（5）尽量使用\n 而不是 endl。使用 endl 会迫使输出流刷新其内部缓冲区。如果该流的确有内部缓冲区，而且又确实不需要每次都刷新它的话，可以在整个循环结束之后写一行刷新语句，这样程序会执行得快很多。

最后一个意见稍微高级一些：

（6）尽量使用标准库中的 copy()和 for_each()，而不是自己手写循环，因为利用标准库的设施，你的代码可以变得更为干净简洁。这里，风格跟美学判断起作用了。在简单的情况下，copy()和 for_each()可以而且确实比手写循环的可读性要强。不过，也只有像本例这样的简单情形才会如此，如果情况稍微复杂一些的话，除非你有一个很好的表达式模板库，否则使用 for_each()来写循环反而会降低代码的可读性，因为原先位于循环体中的代码必须被提到一个仿函数当中才能使用 for_each()。有时候这种提取是件好事，但有时它只会导致混淆晦涩。

之所以说大家的口味可能各不相同，就是这个原因。另外，在本例中我倾向于将原先的手写循环替换成如下的形式：

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

此外，如果你如此使用 copy()，那么原先关于!=、++、end()以及 endl 的问题就不用操心了，因为 copy()已经帮你做了这些事情（当然，我还是假定你并不希望在每输出一个 int 的时候都去刷新输出流，否则你只有手写循环了）。复用如果运用得当的话不但能够改善代码的可读性，而且还可以避开一些陷阱，从而让代码更佳。

你可以更进一步，编写一个基于容器的复制算法，也就是说，施加在整个容器（而不仅仅是迭代器区间）之上的算法。这种做法同样也可以自动纠正 const_iterator 问题。例如：

```
template<class Container, class OutputIterator>
OutputIterator copy(const Container& c, OutputIterator result) {
  return std::copy(c.begin(), c.end(), result);
}
```

这里，我们只需简单地包装 std::copy()，让它对整个容器进行操作，此外由于我们是以 const&来接受容器参数的，因而迭代器自然就是 const_iterator 了。

7

准则： 确保const正确性。特别是不对容器内的元素做任何改动的时候，记得使用const_iterator。
尽量使用!=而不是<来比较两个迭代器。
养成默认情况下使用前缀形式的--和++的习惯，除非你的确需要用到原来的值。
实施复用：尽量复用已有的算法，特别是标准库算法（例如for_each()），而不是手写循环。

接下来我们遇到下面这行代码：

```
cout << v[0];
```

当程序执行这一行的时候，可能会打印出 1。这是因为前面的程序以错误的方式改写了 v[0]所引用的那块内存，只不过，这行代码也许并不会导致程序立即崩溃，真遗憾！

```
v.reserve(100);
assert(v.capacity() == 100);
```

同样，这里的断言表达式当中应该使用>=，而且和前面一样，这也是多余的。

```
cout << v[0];
```

很奇怪！这次的输出结果可能为 0，我们刚刚赋值的 1 神秘失踪了！

为什么？我们假设 reserve(100)确实引发了一次内部缓冲区的重分配（即如果第一次 reserve(2)并没有使内部缓冲区扩大到 100 或更多的话），这时 v 就只会将它确实拥有的那些元素复制到“新家”当中，而问题是实际上 v 认为它内部空空如也（因此不进行任何元素拷贝）！另一方面，新分配的内部缓冲区最初值可能为 0（严格讲不确定），因此就出现了上述情况。

```
v[2] = 3;
v[3] = 4;
// ……
v[99] = 100;
```

毫无疑问，看到如上的代码你可能已经叹着气摇头了。这真是糟糕、糟糕、太糟糕了！但由于标准并不强制 operator[]()进行越界检查，所以在大多数实现上这种代码或许会静悄悄地“正确”运行着，而不会立即导致异常或内存陷阱。

如果这样改写：

```
v.at(2) = 3;
v.at(3) = 4;
// ……
v.at(99) = 100;
```

那么问题就会变得明朗了，因为第一个调用语句就会抛出一个 out_of_range 异常。

```
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
  cout << *i << endl;
}
```

再一次提醒，以上代码什么也不会打印出来，应当考虑将它改写成：

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

再次注意，这种复用自动地解决了!=、前缀++、end()以及 endl 问题，因此程序永远不会在这些方面犯错误。良好的复用通常也会让代码自动变得更快和更安全。

小结

了解size()和capacity()之间的区别，了解operator[]()跟at()之间的区别。如果需要越界检查，请使用at()而不是operator[]()。这么做可以帮助我们节省大量的调试时间。

9

第 2 条　字符串格式化的“动物庄园”之一：sprintf

难度：3

在本条及下一条中，我们对sprintf的是是非非进行了一次奥威尔[1]式的严格考察，并指出为什么说其他替代方案总是（对，总是）比sprintf好。

JG问题

1. 什么是 sprintf？尽可能多地列举出 sprintf 的替代方案。

Guru问题

2. sprintf 的主要优势跟弱点分别是什么？请明确加以说明。

解决方案

“所有动物都是平等的，但其中有些动物比其他动物更‘平等’。”

——乔治·奥威尔，《动物庄园》

1. 什么是 sprintf？尽可能多地列举出 sprintf 的替代方案。

考虑如下的 C 代码，它使用 sprintf 将一个整型值转化为可读的字符串形式，之所以要这样做可能是为了将一个整型输出到报表或者打印到 GUI 窗口上：

```
// 示例2-1:在C里面使用sprintf来字符串化某些数据
// PrettyFormat()接受一个整型为参数，将它格式化并
// 放入给定的输出缓冲区当中。
// 出于格式化的考虑，格式化的结果必须至少为4个字符宽。
//
void PrettyFormat(int i, char* buf) {
 // 代码就这些，优雅、简洁
 sprintf(buf, "%4d", i);
}
```

大奖问题是，在 C++中应该如何完成这件事情呢？ 10

呃，好吧，其实问题不该这样问，毕竟示例 2-1 也是合法的 C++代码。真正的大奖问题是：抛开 C++标准[C++03]从 C 标准[C99]那儿承袭来的桎梏和局限性（如果它们的确是桎梏的话），是不是有办法借助于 C++中的类和模板等特性来将这件事做得更好呢？

1. 乔治·奥威尔（1903—1950），英国著名作家，作品有《动物庄园》和《一九八四》。他的作品中创造的“冷战”、“老大哥”等词和许多名言，已成为英语中的日常语汇。——编者注

问题在这里开始变得有趣起来，因为实现这一目的，至少有不下四种截然不同的、直截了当的标准做法，示例 2-1 是其中的第一种。其中任一种都提供了在清晰性、类型安全性、运行时安全性以及效率之间的权衡。此外，套用乔治·奥威尔小说中的那只修正主义的猪的名言："所有这四种选择都是标准的，但其中有些比其他选择要'更标准'一些。"而且，说得更严重一些，它们并非全都基于同一个标准。它们分别是（后面将按照此顺序讨论）：

- sprintf [C99, C++03]
- snprintf [C99]
- std::stringstream [C++03]
- std::strstream [C++03]

除此之外，还有另一个"目前虽不合标准但很有希望成为标准"的替代方案，好像嫌手头的方案还不够多似的，它就是：

- **boost::lexical_cast** [Boost]

boost::lexical_cast 主要用在不需要任何特殊格式化的简单转换当中。

好了，闲话少说，言归正传。

sprintf()的悲与乐

2. sprintf 的主要优势跟弱点分别是什么？请明确加以说明。

示例 2-1 中的代码只是使用 sprintf 的众多可能的方式中的一种。我们用示例 2-1 来引发下文的讨论，不过不要过分依赖于这个简单得只有一行代码的 PrettyFormat 函数。要记住我们的大方向：我们的兴趣在于通常情况下如何将非字符串的值格式化为字符串形式，或许在实际编码当中，我们的做法是在不断变化和改进着的，而不像示例 2-1 当中的一行简单代码那样。

下面我们将更详细地分析 sprintf()，并列出其中存在的主要问题。sprintf()有两个主要的优势，还有三个明显的缺陷。其中两个优势为：

议题#1：易用性与清晰性。一旦你学会了 sprintf 的常用格式化标志以及它们的各种组合，其使用就会变得简洁明了，没有任何拐弯抹角之处。使用 sprintf 的代码明白无误地说明了它正在做的事情。因此 printf 家族在大多数文本格式化场合下是很难有功能能够与之匹敌的（确实，我们中的大部分人有时仍然免不了需要去查寻一些不常用的标志，不过它们毕竟用得很少）。

11

议题#2：效率最佳（能够直接利用现有的缓冲区）。通过使用 sprintf 将结果直接放入一个已有的缓冲区中，PrettyFormat()将不用牵涉任何动态内存分配或者其他额外的幕后操作就能完成任务。将一块已分配好用于存放输出结果的缓冲区传递给 PrettyFormat()，后者负责将格式化的结果直接写入这块缓冲区。

告诫：当然，现在也不必过分在乎效率，因为你的应用程序或许根本就不会在意这一点效率差别。永远不要过早进行优化，只有当时间测试显示确实有必要时才去进行优化。而且，遇到这种情况的时候，永远不要忘记，效率是以牺牲内存管理封装性而换取的。议题#2 等于是在说："你自己去管理内存，"不过别忘了，这句话换一种说法就是"你得自己去管理内存"！

只可惜，正如大多数使用 sprintf 的程序员所知道的，情况还远远不止这些。sprintf 同样存在一些显著的缺陷：

议题#3：长度安全性。sprintf 是引起缓冲区溢出错误的原因之一，如果目标缓冲区碰巧不够大，装不下整个输出结果的话，就会发生缓冲区溢出[1]。例如，考虑如下的代码：

```
char smallBuf[5];
int value = 42;
PrettyFormat(value, buf);                    // 呃……隐患
assert(value == 42);
```

本例中，42 恰好足够小，以至于 5 字节大小的结果“ 42\0”恰巧能够放在 smallBuf 中。然而，设想某一天代码改成了这样：

```
char smallBuf[5];
int value = 12108642;
PrettyFormat(value, buf);                    // 哦!
assert(value == 12108642);                   // 这个断言很可能会失败!
```

这会导致 smallBuf 尾部之后的区域也被改写，而倘若编译器恰巧让 value（在内存中）紧跟在 smallBuf 之后的话，被改写的区域就是 value 值本身占用的空间了！

12

我们无法轻易地改善示例 2-1 的安全性。的确，我们可以让 PrettyFormat()接受缓冲区的长度并对 sprintf()的返回值进行检查，但这等于是事后诸葛亮。具体做法如下：

```
// 糟糕的主意：丝毫没有改观。
//
void PrettyFormat(int i, char* buf, int buflen) {
  if(buflen <= sprintf(buf, "%4d", i)) {    // 并不比以前好
    // ……现在情况如何呢？既然在这里问题被侦测出来了，那么这就
    // 意味着问题已经发生了，换句话说该被破坏的内存已经被破坏了
    }
}
```

对于这个问题，根本没有解决方案。当错误被侦测出来时，内存已然被破坏，我们已经在不该写的地方写下了一些字节，如果情况糟糕的话，程序甚至根本没机会运行到报错代码处[2]。

---

1. 一个初学者易犯的错误就是依赖于宽度指示，这里是 4，这种做法是不行的，因为宽度指示只表示最小宽度，不表示最大宽度。
2. 注意，在某些情况下，至少理论上你可以缓解缓冲区长度问题，做法是在运行时创建自己的格式。之所以说“理论上”，是因为这种做法通常是不现实的，代码总是显得晦涩而脆弱。正如 Bjarne Stroustrup 在[Stroustrup99]中所提到的一个类似的情况：

   我并不认为将下面这个“专家级”的替代方案介绍给新手是个好主意：

```
char fmt[10];
// 创建一个格式字符串：简单地使用%s可能会导致缓冲区溢出
sprintf(fmt,"%%%ds",max-1);
// 至多将max-1个字符读入name中
scanf(fmt,name);
```

议题#4：类型安全性。对于 sprintf 来说，类型错误就意味着运行时错误，而非编译期错误，更可怕的是这些类型错误甚至根本就不会表现出来。printf 家族使用 C 的可变参数列表，C 编译期通常并不检查这类实参列表的类型[1]。几乎每个 C 程序员都曾在一些微妙的或者不那么微妙的情况下发现他们搞错了格式字符串，这类错误总是再频繁不过地发生着，譬如在熬夜调试之后，试图重现某个关键客户遇到的神秘崩溃问题时。

诚然，示例 2-1 中的代码是如此简单，以至于只要清楚地知道我们只是将一个 int 传给 sprintf，
13 就可能足够简单地维护它。不过，即便如此，事情仍然可能出现纰漏，设想你的手指一不小心按错了键，这类情况并不罕见。例如，在大多数键盘上，c 键跟 d 键是相邻的，所以我们可能一不小心把 d 错打成了 c，结果就成了这样：

```
sprintf(buf, "%4c", i);          // 哦！
```

这会导致输出结果为字符而不是数字，这种情况下我们或许很快就能意识到错误所在，因为 sprintf 会一声不吭地将 i 的第一个字节解释为一个 char 值。此外，s 键也跟 d 键相邻，因此如果我们错误地写成了：

```
sprintf(buf, "%4s", i);          // 糟糕！
```

如果情况是这样的话，或许我们同样能够很快反应过来，因为这么做很可能会令程序立即崩溃或至少偶发性地崩溃。因为这时 sprintf 会不加提示地将 i 解释为指向字符串的指针，并欣然地顺着这个指针所指的方向去寻找一个实际上并不存在的字符串，实际上，这个指针可能指向内存中的任何位置。

不过，下面这种情况可就微妙了，假设我们将 d 错打成了 ld，会出现什么情况呢？

```
sprintf(buf, "%4ld", i);         // 一个微妙的错误
```

若是这种情况的话，给出的格式字符串就等于是在告诉 sprintf，给它的是 long int，而实际上给的却是 int。这同样也是糟糕的 C 代码，不过，问题是它不仅不会以编译期错误的形式表现出来，甚至不会导致运行时错误。在许多流行的平台上，程序的运行结果仍然会跟以前一样，因为 int 在许多流行平台上碰巧跟 long int 具有相同的大小和内存布局。因而你也许一直都不会注意到这个潜在的问题，直到某一天将代码移植到某个平台上，该平台上的 int 跟 long int 具有不同的大小，这时才发现这个问题，甚至就连这个时候，程序可能也并不总是产生错误的结果或立即崩溃。

最后，考虑一个与此有关的问题。

议题#5：模板亲和性。很难将 sprintf 放在一个模板当中。考虑如下的代码：

```
template<typename T>
void PrettyFormat(T value, char* buf) {
  sprintf(buf, "%/* 这里应该写些什么呢？ */", value);
}
```

1. 使用类似于 lint 的工具有助于捕获这类错误。

你所能做到的最好的（最糟的？）就是声明一个主模板，并为所有那些与 sprintf 兼容的类型分别提供对应的特化版本：

```
// 不算好点子：  一个东拼西凑出来的PrettyFormat
//
template<typename T>
void PrettyFormat(T value, char* buf);      // 注意：主模板只有声明，没有定义

template<> void PrettyFormat<int>(int value, char* buf) {
  sprintf(buf, "%d", value);
}
template<> void PrettyFormat<char>(char value, char* buf) {
  sprintf(buf, "%c", value);
}
//……还有其他特化版本，呃……
```

14

总的来说，sprintf 是这样的：

	sprintf
标准吗	是: [C90], [C++03], [C99]
易用吗？代码清晰明确吗	是
高效吗？没有额外的内存分配吗	是
长度安全吗	否
类型安全吗	否
可用在模板当中吗	否

下一条我们将会考虑其他的解决方案，它们是在以上这些考虑因素之中进行取舍的结果。

15

第 3 条 字符串格式化的“动物庄园”之二：标准的（或极度优雅的）替代方案

难度：6

对sprintf问题的奥威尔式的严格考察，最终以我们对snprintf、std::stringstream、std::strstream以及非标准但极度优雅的boost::lexical_cast的一番对比分析结束。

Guru问题

1. 比较下面这些替代方案的优点和弱点，使用第 2 条中的分析和示例代码。

（a）**snprintf**

（b）**std::stringstream**

（c）**std::strstream**

（d）**boost::lexical_cast**

解决方案

替代方案#1：snprintf

1. 比较下面这些替代方案的优点和弱点，使用第 2 条中的分析和示例代码。

（a）**snprintf**

在所有的选择当中，与 sprintf 最相近的选择当然是 snprintf 了。snprintf 只是在 sprintf 上增加了一项功能，不过是一项重要功能，即用户可以给出输出缓冲区的最大长度，从而避免缓冲区溢出。当然，如果缓冲区太小的话，输出结果就会被截断。

长久以来，在大多数 C 实现上，snprintf 都是作为一个非标准的扩展存在的。随着 C99 标准的颁布[C99]，snprintf 终于浮上台面而成为合法功能，目前 snprintf 已经是 C99 标准中的正式一员。不过，除非你的编译器是符合 C99 标准的，否则可能仍然必须使用供应商提供的非标准扩展，如_snprintf。

16

坦白地说，早该使用 snprintf 来取代 sprintf 了，即使在 snprintf 还没有标准化之前。大多数良好的编码标准都不推荐你使用像 sprintf 这样的不检查长度的函数，而且该原则是很有道理的。使用不做检查的 sprintf 长久以来常引起一些声名狼藉的常见问题，它通常会导致程序崩溃[1]，尤

1. 这是一个实实在在的问题，而且不仅是 sprintf()，C 标准库当中的其他任何不进行（缓冲区）长度检查的调用都存在这个问题。试着用 Google 搜索“strcpy”和“buffer overflow”，你就会发现这一周来 strcpy 又闯了哪些祸。

其会导致安全脆弱问题[1]。

借助于 snprintf，我们就可以正确编写刚才一直试图实现的带长度检查的 PrettyFormat()版本。

```
// 示例3-1: 在C中使用snprintf来字符串化某些数据
//
void PrettyFormat(int i, char* buf, int buflen) {
 // 这就是代码，简洁优雅，关键是比以前要安全得多:
 snprintf(buf, buflen, "%4d", i);
}
```

注意，即便这样做了，仍然还存在另一种出错的可能，即调用者将缓冲区长度搞错了。这意味着跟那些具有资源管理功能的替代方案相比，snprintf 还算不上百分之百地杜绝缓冲区溢出可能性，不过跟 sprintf 相比它显然要安全多了，在“长度是否安全？”这个问题上应该算是合格的。使用 sprintf 没有合适的途径来绝对避免缓冲区溢出，而通过 snprintf，我们则可以（很大程度上）杜绝缓冲区溢出。

注意，snprintf 的一些标准化之前版本的行为稍有不同。尤其是在一个主要实现中，如果输出结果填满或者大于缓冲区容量，缓冲区里的串就不会以'\0'结尾。这种情况下，我们的 PrettyFormat()函数就得稍作调整以应付这种非标准的行为：

```
// 在C中使用一个并不十分遵从C99标准的_snprintf来将数据字符串化
//
void PrettyFormat(int i, char* buf, int buflen) {
 // 这里是代码，简洁优雅，而且安全得多
 if(buflen > 0) {
    _snprintf(buf, buflen-1, "%4d", i);
    buf[buflen-1] = '\0';
 }
}
```


在其他任何方面，sprintf 和 snprintf 都是一样的。综上所述，snprintf 跟 sprintf 的比较如表 3-1 所示。

表 3-1　snprintf 与 sprintf 的比较

	snprintf	sprintf
标准吗	是：仅[C99]，不过也可能进入 C++0x	是：[C90]，[C++03]，[C99]
易用吗，代码清晰明确吗	是	是
高效吗，无额外的内存分配吗	是	是
长度安全吗	是	否
类型安全吗	否	否
可用于模板之中吗	否	否

1. 例如，以前有很长一段时间，许多恶意网站都流行一种做法，就是向访问者的浏览器发送非常长的 URL 字符串，长到大于浏览器的内部 URL 缓冲区，从而造成浏览器的缓冲区溢出，进而造成浏览器程序崩溃。那些不对 URL 串长度进行检查而是直接将它们拷贝到内部定长缓冲区中的浏览器，遇到这种恶意攻击的结果，就是改写超过缓冲区尾部之后的内存，通常这也就是覆盖原本的数据而已，不过在黑客精心安排下也可以达到改写代码段的目的，借机写入一些会执行的恶意代码。我们简直没法想像“古往今来”有多少软件在使用着不加检查的调用。

从这些比较当中，我们可以给出如下的建议：

准则： 永远不要使用sprintf。

如果你真的决定使用 C 的 stdio 设施的话，一定要记住，使用那些进行长度检查的函数，如 snprintf。即便在你的编译器上它们只是作为非标准扩展存在，也得使用它们，因为使用它们没有任何损失，还能够带来实实在在的好处。

我曾在 C++大会上将该主题作为演讲稿的材料，一开始我便惊讶地发现，通常在一场大会的所有到场人员当中只有百分之十的人听说过 snprintf。然而，几乎每次，当我问到关于 sprintf 在实际项目当中导致的问题时，总会有人立即举手，描述他们最近在项目当中发现一些缓冲区溢出 bug，而当他们在整个项目中将 sprintf 全部替换为 snprintf 之后再去进行测试，发现不但这些 bug 消失了，就连其他一些早就报告了的、已经在 bug 队列里面呆了很长时间却一直没人能够解决的神秘 bug 也随之消失了。

结论，正如我一直所说的，是永远不要使用 sprintf。

替代方案#2：std::stringstream

（b）**std::stringstream**

C++中用于字符串化的最常见设施就是 stringstream 这一族的类了。示例 3-1 如果用 ostringstream 来替代 sprintf 的话看起来就会像这样：

18

```
// 示例3-2：在C++中进行字符串化，使用ostringstream
//
void PrettyFormat(int i, string& s) {
 // 不如原先的简洁优雅
 ostringstream temp;
 temp << setw(4) << i;
 s = temp.str();
}
```

相对于 sprintf 来说，stringstream 具有一些优点，但同时也有缺点。在 sprintf 光芒四射的那些地方，stringstream 显得并不那么出色。

*议题#1：易用性和清晰性。*使用 stringstream 不仅让原先的一行代码变成了三行，而且我们还得引入一个临时变量。使用 stringstream 的做法有几个优势，不过代码的清晰性并非其中之一。这并不是说像 setw(4)这样的流操纵子难于学习，实际上它们跟 sprintf 的格式化标志一样易学，只不过前者通常更为笨拙冗长一些。我发现那些到处“点缀”着像<<setprecision(9)和<<setw(14)这样的长名字的代码会难于阅读（我是说，跟%14.9 这种格式化字符串相比），即便所有的操纵子都整齐排列也无济于事。

议题#2：效率（能否直接利用现有缓冲区）。stringstream 会自己另外分配一份单独的缓冲区来存放结果，另外还需要使用一些辅助性的对象，通常所有这些都意味着需要进行额外的内存分配。我在两个当前流行的编译器上测试了示例 3-2 的代码，同时让::operator new 统计总共的分配次数。结果发现在某个平台上有两次动态内存分配，另一个平台上则是三次。

而在 sprintf 一筹莫展的那些地方，stringstream 则大显身手。

议题#3：长度安全性。stringstream 内部的 basic_stringbuf 缓冲区类会根据需要自动增长，以便容纳需要存放的数据。

议题#4：类型安全性。使用 operator<<和重载决议，即便是对于那些提供了自己的流插入操作符的自定义流类型，也总能够实现类型安全性。不会因为类型不符而导致一些神秘的运行时错误。

议题#5：模板亲和性。既然编译器会自动调用正确的 operator<<，那么将 PrettyFormat 泛化为可接受任何类型的数据应当是举手之劳：

```
template<typename T>
void PrettyFormat(T value, string& s) {
  ostringstream temp;
  temp << setw(4) << value;
  s = temp.str();
}
```

综上所述，stringstream 跟 sprintf 的比较如表 3-2 所示。

19

表 3-2　stringstream 与 sprintf 的比较

	stringstream	sprintf
标准吗	是：[C++03]	是：[C90]，[C++03]，[C99]
易用吗，代码清晰明确吗	否	是
高效吗，无额外内存分配吗	否	是
长度安全吗	是	否
类型安全吗	是	否
可用于模板之中吗	是	否

替代方案#3：std::strstream

（c）**std::strstream**

不管这种说法公平与否，strstream 都是要被遗弃的。由于[C++03]标准将它标明为 deprecated（不赞成的），因而优秀的 C++书籍顶多也只是略微提及一下（见[Josuttis99]的第 649 页），大多数则是根本不提（见[Stroustrup00]），甚至明确地表态说不会讨论这方面的内容，因为 strstream 是官方规定的“替补”（见[Langer00]的第 587 页）。标准委员会觉得 stringstream 可以取代 strstream，因为 stringstream 更好地封装了内存管理，所以他们将 strstream 标明为 deprecated，

然而 strstream 仍然还是标准的法定成员，任何符合 C++标准的实现都必须提供它[1]。

由于 strstream 仍然是标准的，所以为了完整起见这里还是提一下它。碰巧它也的确提供了一些优点。使用 strstream 的话，示例 3-1 看起来就会像这样：

```
// 示例3-3: 在C++中使用ostrstream进行字符串化
//
void PrettyFormat(int i, char* buf, int buflen) {
 // 不算太差，不过别忘了最后还要输出结束符
 ostrstream temp(buf, buflen);
 temp << setw(4) << i << ends;
}
```

20

*议题#1：易用性和清晰性。*strstream 在易用性跟代码的清晰性方面略逊 stringstream 一筹。两者都要求建立一个临时对象，不过 strstream 要求你记得手动输出一个结束符来结束字符串，这除了令人感觉有点不愉快之外，还有点危险，因为如果一不小心忘记了，同时在读取结果串的时候又期望该串是以'\0'字符结尾的话，你就面临着读取超过结果串末尾之后的内存数据的危险，而就算 sprintf 也没这么脆弱，它总是会在结果串的末尾加上结束符。不过，按照示例 3-3 所展示的方式那样使用 strstream 至少有一个优点，即无需在最后调用 c_str()来获取结果串。（当然，如果让 strstream 创建自己的缓冲区，其内存只是部分封装的，你除了得在最后调用.str()来将其中的结果串取出来之外，还得加上一次.freeze(false)调用，否则 strstreambuf 在析构的时候是不会释放内存的。）

*议题#2：效率（能否直接利用现有缓冲区）。*我们只需在创建 ostrstream 对象的时候传递一个指向现有缓冲区的指针，就可以避免任何额外的内存分配，ostrstream 会将它的结果直接输出到该缓冲区当中。这跟 stringstream 相比是一个非常重要的区别，在能否将结果串直接输出到现有的目标缓冲区（从而避免额外的内存分配）这个问题上，stringstream 根本无法与 strstream 比拟[2]。当然，如果你并没有现成可利用的缓冲区，ostrstream 也可以使用自己动态分配的缓冲区，你只需调用它的默认构造函数即可[3]。确实，strstream 是我们所讨论的所有可选方案当中惟一能

1. 从理论以及从实际上说，deprecated 究竟意味着什么呢？当用在标准当中时，带有 deprecated 标记的特性表示，标准委员会警告你该特性可能会在将来的任何时候消失，很可能就是在下一代标准颁布的时候。deprecate 某个特性就意味着以一种标准化的方式来给予用户善意的劝阻，在不将某个特性立即移除的前提下，这种做法是标准委员会所能做出的最为有力的劝阻。而实际上，即便想要将那些最不被赞成使用的特性从标准中移除也是相当难的，因为一旦某个特性出现在了标准当中，人们就会编写出依赖于该特性的代码，而标准委员会里谁都不愿意破坏向后兼容性。因而即便某个特性被移除了，标准的实现商们通常仍然会继续提供它们，因为他们也不想破坏向后兼容性。因而通常来说标明为 deprecated 的特性从来也没有真正地从标准中除掉过。例如，Fortran 标准中仍然残留着十余年前就标明为 deprecated 的特性。
2. stringstream 确实也提供了一个能够接受一个 string&参数的构造函数，只不过它只是简单地对该 string 参数的内容进行复制，而不是直接将它当作工作区。
3. 在表 3-5 的性能测试中，strstream 在两个平台（Borland C++ 5.5.1 和 Visual C++ 7）上皆出人意料地表现出了较差的性能。原因似乎是在这些实现上每次调用示例 3-3 的 PrettyFormat()时不知何故总会进行一些内存分配（尽管就这两个平台而言，利用现有缓冲区（示例 3-3）跟任由 strstream 自己分配缓冲区相比仍然是前者进行的分配次数较少）。而在其他环境中，正如我们所预期的，这种做法没有任何内存分配。

够提供这种选择自由的方案。

议题#3：长度安全性。ostrstream 内部的 strstreambuf 缓冲区会自动检查它的长度以确保不会写超过给定缓冲区之外的内存区域。而如果我们使用的是一个默认构造的 ostrstream 对象的话，其内部的 strstreambuf 缓冲区就会根据需要自动增长以容纳有待存储的值。

议题#4：类型安全性。strstream 跟 stringstream 一样，完全是类型安全的。

议题#5：模板亲和性。完全可以！正如 stringstream 一样。例如：

21

```
template<typename T>
void PrettyFormat(T value, char* buf, int buflen) {
  ostrstream temp(buf, buflen);
  temp << setw(4) << value << ends;
}
```

总之，strstream 与 sprintf 的比较结果如表 3-3 所示。

表 3-3　strstream 与 sprintf 的比较

	strstream	sprintf
标准吗	是：[C++03] 不过标明为 deprecated 了	是：[C90]， [C++03]，[C99]
易用吗，代码清晰明确吗	否	是
高效吗，无额外内存分配吗	是	是
长度安全吗	是	否
类型安全吗	是	否
可用于模板之中吗	是	否

呃……看到一个已被“打入冷宫”的设施在比较中表现如此良好的确让人有几分尴尬，不过有时候现实就是如此。

替代方案#4：boost::lexical_cast

（d）boost::lexical_cast

如果你还没有接触过[Boost]的话，我的建议是，马上去研究它！Boost 是 C++的一个开源库，主要由 C++标准委员会成员编写。其代码经过严格的同行评审，并由专家编写的，遵从 C++标准库的风格，同样，其中设施的明确意图就是作为下一代 C++标准库的潜在候选子库，因此花些时间去了解它们是完全值得的。此外，从今天开始你就可以完全免费地使用它们了。

Boost 库中提供的设施之一就是 boost::lexical_cast，它是 stringstream 的一个易用的包装类。此外 Boost 中还包括一些其他更为华丽和重要的设施，它们同样在内部借助于流来实现，并提供了更为 sprintf 式的格式化选择，其中尤为突出的要数 boost::format。由于 Kevlin Henney 写的 Boost 代码实在是太简洁优雅了，因此我可以完完整整地将它们罗列在下面（删掉了一些为旧的编译器所做的 workaround），所以虽说它目前还未被标准化，我也乐于将它介绍给读者：

```
template<typename Target, typename Source>
Target lexical_cast(Source arg) {
  std::stringstream interpreter;
  Target result;
  if(!(interpreter << arg) || !(interpreter >> result) || !(interpreter >> std::ws).eof())
    throw bad_lexical_cast();

  return result;
}
```

22

注意，lexical_cast 的意图并非是想要成为 sprintf 的直接竞争者。实际上，sprintf 比 lexical_cast 更为通用，而 lexical_cast 的目的只是为了将数据从一个可流化的类型转换为另一个可流化的类型，因而它与 C 中的 atoi 等转换函数以及非标准但广泛使用的 itoa 等函数的竞争更为直接一些，然而，lexical_cast 与我们当前讨论的主题又是如此地接近，以至于不提及它就明显是个疏忽了。

下面就是使用 lexical_cast 来改造示例 3-1 后的情形：

```
// 示例3-4: 在C++中使用boost::lexical_cast进行字符串化
//
void PrettyFormat(int i, string& s) {
  // 如果这确实恰是你所需要的，那么可以说这是目前最为简洁优雅的做法了
  s = lexical_cast<string>(i);
}
```

议题#1：易用性和清晰性。在所有这些例子当中，使用 lexical_cast 的代码最为直接地表达了实际意图。

议题#2：效率（能否直接利用现有缓冲区）。由于 lexical_cast 使用的是 stringstream，因此毫不奇怪它需要至少跟 stringstream 一样多的内存分配次数。在我测试的一个平台上，示例 3-4 比示例 3-2 中直接使用 stringstream 的版本多进行了一次内存分配，而在另一个平台上则没有多出。

跟 stringstream 一样，在长度安全性、类型安全性以及模板亲和性这些方面，lexical_cast 也有优秀的表现。

总之，lexical_cast 跟 sprintf 的比较如表 3-4 所示。

表 3-4 lexical_cast 与 sprintf 的比较

	lexical_cast	sprintf
标准吗	否：有可能成为[C++0x]的候选	是：[C90]，[C++03]，[C99]
易用吗，代码清晰明确吗	是	是
高效吗，无额外内存分配吗	否	是
长度安全吗	是	否
类型安全吗	是	否
可用于模板之中吗	是	否

23

小结

到目前为止，还有一些问题是我们未曾详细讨论过的，例如，这里所讨论的所有字符串格式化都是针对基于 char 类型的窄字符串的，而并没有涉及宽字符串。我们也考察了让 sprintf、snprintf、strstream 直接利用现有缓冲区而带来的性能提升，然而“你自己去管理内存”的另一面就是“你得自己去管理内存”，因而 stringstream、strstream 还有 lexical_cast 提供的封装得更好的内存管理可能相当吸引你。（这里并没有打错字，strstream 的确是个双面手，具体要看你如何使用它。）

另外，还有一些我们并没有详细讨论的非标准方案。我选择将 boost::lexical_cast 作为非标准方案的代表是因为它简洁优雅，不过即便是在 Boost 当中也存在着更为完备更为重量级的方案，特别值得注意的是 boost::format，它建立在与这儿提到的 stringstream 和 strstream 技术类似的方法基础之上，提供了更自动化的能力来支持 sprintf 式的格式化。

将这些放到一起，我们可以得到综合比较的表 3-5。考虑到我们在判断每种方案的优劣时应考虑的那些方面，很显然这些方案当中没有哪个是在任何情况下都合适的。

表 3-5　C/C++字符串格式化方案

	sprintf	snprintf	stringstream	strstream	boost::lexical_cast
	标准吗？（— = n/a）				
[C90]	是	否（常见扩展）	—	—	—
[C++03]	是	否（常见扩展）	是	是（但是 deprecated）	否
[C99]	是	是	—	—	—
C++0x（推测）	是	很可能	是	很可能（也可能仍 deprecated）	可能
	可用性考虑				
易用吗，代码清晰明确吗	是	是	否	否	是
高效吗，无额外内存分配吗	是	是	否	是	否
长度安全吗	否	是	是	是	是
类型安全吗	否	否	是	是	是
是否可用于模板之中	否	否	是	是	是
	耗时取样，以 sprintf 的耗时为基准[1]				
Borland C++ 5.5.1/Windows	1.0	1.0	12.6	8.1	19.7
Gnu g++ 2.95.2/Cygwin+Windows	1.0	—	—	2.0	—
Microsoft VC7/Windows	1.0	1.0	13.2	9.0	19.2
Rogue Wave 2.1.1/ SunPro 5.3/SunOS 5.7	1.0	1.1	8.7	4.7	16.5
Rogue Wave 2.2.1/HP aCC 3.30/HP-UX 11.00	1.0	1.0	7.9	3.9	9.9

1. 测试程序将相应的示例代码调用 1 000 000 次，运行测试程序 3 次，取所耗时间的平均值。随着编译器版本以及编译选项设置的不同，测试结果可能会有所不同。

在表 3-5 的基础上，我们可以得出如下准则（表 3-6 中同样也进行了一番总结）：

- 如果你所要做的只是将一个值转换为一个字符串（甚至于其他任何类型）的话：尽量默认使用 boost::lexical_cast。
- 如果想进行简单的格式化，或者需要支持宽字符串，或者想要让进行格式化的代码能够用在模板当中：尽量使用 stringstream 或 strstream。跟 snprintf 相比，使用 stringstream/strstream 的代码会比较冗长，且难于理解，不过对于简单的格式化任务来说，情况不会太糟。
- 如果想要进行一些更为复杂的格式化任务，同时并不需要宽字符串支持，也不想让代码用在模板当中的话：尽量使用 snprintf。这是 C 的做法，但并不意味着 C++程序员就不能用它。
- 仅当实际的性能测试显示那些较好的替代方案在你代码中某个特定的地方确实都会造成瓶颈时，只在这些特定地点改用 strstream 或 snprintf，具体用哪个则要看谁更适合。

24

- 永远不要使用 sprintf。

表 3-6 准则小结

	默认情况下，当效率不是关键时	如果效率确实低到成问题的话（这必须是根据性能分析结果得出的结论，而不是臆测）
如果想要做的只是将数据转换为其字符串表示的话	boost::lexical_cast	std::strstream 或 snprintf
较为简单的格式化任务，或者需要宽字符串支持，或者需要让格式化代码能够用于模板当中	std::stringstream 或 std::strstream	std::strstream 或 snprintf
较为复杂的格式化任务，而且不需要宽字符串支持，也不需要将负责格式化任务的代码置于模板当中	snprintf	snprintf

25

最后，我还想针对被“贬”的 strstream 说一点：strstream 提供了一个有意思的特性组合，尤其是它是所有方案当中惟一一个允许你自由选择是否自己进行内存管理的；使用 strstream，如果你不想自己管理内存的话，你大可以把担子交给它，让它来（不完全地）封装内存管理。从技术上来说，它的惟一缺陷在于使用起来过于脆弱，这是因为它需要你手动输出结束符来结束一个字符串，还因为它的内存管理方式。它的另外一个软肋则是它在公众当中的坏名声。由于它曾经被束之高阁，没有人会再去过多使用它，而且你还得留神着标准委员会以及你的编译器/库供应商们，因为他们以后可能真的会将这一设施移除，虽然这种可能性微乎其微，但并不等于零。

看到一个被“贬谪”的特性如此出风头确实令人有几分尴尬。虽说某种特定的“动物”可能会具有一些独一无二的优势，但是即便是在标准中，有些动物仍然要比其他动物更“平等”。

26

第 4 条　标准库成员函数

难度：5

复用是好的，但你是否总能够将标准库设施复用于自身呢？下面是一个可能会让你吃惊的例子，你将会看到，标准库中有一个设施可以可移植地复用到你的任何代码中，只要你愿意。然而，它却不能可移植地复用到标准库自身。

JG问题

1. 什么是 std::mem_fun？你什么时候会去使用它？给出一个例子。

Guru问题

2. 假设在下面代码中的/*...*/处放入恰当的模板参数，这行代码会成为合法且可移植的 C++ 代码吗？请说出理由。

```
std::mem_fun</*...*/>(&(std::vector<int>::clear))
```

解决方案

mem_fun

1. 什么是 std::mem_fun？你什么时候会去使用它？给出一个例子。

标准库里面的 mem_fun 是一个适配器（adapter）类，它能够将成员函数适配为所谓的仿函数（functor），从而可被标准库算法以及其他正常情况下只使用自由函数的代码所使用。

例如，假设有如下的代码：

```
class Employee {
public:
  int DoStandardRaise() { /*...*/ }
  //……
};

int GiveStandardRaise(Employee& e) {
  return e.DoStandardRaise();
}

std::vector<Employee> emps;
```

下面这种用法可能是我们司空见惯了的：

```
std::for_each(emps.begin(), emps.end(), &GiveStandardRaise);
```

27

但是，如果 GiveStandardRaise()并不存在，或者由于某些原因需要直接去调用 Employee 中对应的成员函数的话，我们该怎么办呢？答案是我们可以这么写：

```
std::vector<Employee> emps;
```

```
std::for_each(emps.begin(), emps.end(),
              std::mem_fun_ref(&Employee::DoStandardRaise));
```

mem_fun_ref 末尾的_ref 是由一些历史性的古怪癖好造成的。在写这类代码的时候，如果你的容器是老式的，其中包含的是对象的话，你就应当记住使用 mem_fun_ref，因为 for_each 操纵的将会是容器中对象的引用。而倘若容器中包含的是指向对象的指针，你就应该使用 mem_fun：

```
std::vector<Employee*> emp_ptrs;
std::for_each(emp_ptrs.begin(), emp_ptrs.end(),
              std::mem_fun(&Employee::DoStandardRaise));
```

你可能已经注意到了，我在示例中展示的成员函数皆是无参的。对于那些接受一个参数的成员函数，可以利用 std::bind...()辅助函数，使用原则跟 mem_fun 一样。然而遗憾的是，这种做法不适用于那些接受两个或多个参数的函数。不过话说回来，这并不能抹杀其用处。

这就是 mem_fun。mem_fun 的这些特质给我们带来了一些尴尬的事情。

使用 mem_fun，但不要将它用在标准库自己身上

2. 假设在下面代码中的/*...*/处放入恰当的模板参数，这行代码会成为合法且可移植的 C++代码吗？请说出理由。

```
std::mem_fun</*...*/>(&(std::vector<int>::clear))
```

首先我们注意到，其实/*...*/处根本没必要放置任何模板参数。我故意这样问你是因为在写作本书的时候仍然还有部分流行的编译器不能正确地推导出模板参数。在这些编译器上，你必须手动给出模板参数。取决于你的标准库实现，你的代码看起来可能像这样：

```
std::mem_fun<void, std::vector<int, std::allocator<int> > >(&(std::vector<int>::clear));
```

随着时间的推移，这种局限性将会慢慢消失，并且编译器也可以让你放心地省略模板参数。

你可能想知道为什么我刚才要说“取决于你的标准库的实现”。毕竟，std::vector<int>::clear()的函数签名是无参、返回 void，不是吗？标准中是这么说的，对吗？

28 也许你错了！这里我们触摸到了问题的症结所在。

C++标准里面关于标准库的部分在描述某些成员函数的实现时故意留了一些余地。尤其是下面这句话。

- 一个具有默认参数的成员函数签名可以被“两个或多个具有等价行为的成员函数签名”所替代。
- 成员函数签名可以具有额外的默认参数。

是的，上面的第二句话就是问题所在，即那些可有可无、若隐若现的额外参数（让我们简称它们为“皮卡布”[1]参数）就是肇事者。

大多数时候，那些由实现定义的、额外的默认“皮卡布”参数根本不会引起人们的注意。例

1. peekaboo 的音译，peekaboo 是一种类似于捉迷藏的游戏，这里被作者用来戏称那些可有可无的默认形参。——译者注

如，当你调用一个成员函数的时候，那些“皮卡布”参数会绑定到它们的默认值，因而平常你根本不用去关心标准库是否为这些函数添加了一些额外的形参。然而遗憾的是，当你不得不知道某个成员函数的确切签名时，这些可能的额外形参就成了大问题！譬如在你使用 mem_fun 的时候。注意，即便你的编译器能够正确地推导出模板实参也无济于事，这是因为两个潜在问题。

- 如果有问题的成员函数实际上具有一个带默认值的形参，而你又不期望出现这个形参的话，你就得使用像 std::bind2nd 这样的设施来对付它（要想知道更多关于标准约束器（binder）辅助设施的描述，请参考[Josuttis99]）。当然，这么做了之后，如果你将代码移植到另一个平台上，而该平台上的标准库中相应的成员函数却具有一个不同类型的额外形参，或者根本没有额外的形参，那么你的代码同样无法工作。不过话说回来，你的代码本来就是不可移植的，不是吗？
- 如果有问题的成员函数实际上具有两个或多个形参（即便它们都是有默认值的），就根本没法将 mem_fun 用在自身上。很不愉快，是吗？不过还是那句话，你的代码反正是不可移植的，对不对？

不过实际上问题也许并没有那么糟。我不知道库实现者是否普遍都会利用这种自由度来添加额外的形参，或者是否可能以后这么做。我知道的是，只要他们不这么做，你在实际中就不会遇到这些难题。

然而遗憾的是，这还不是故事的结尾。最后，让我们考虑一个更为一般性的结果。

使用指向成员函数的指针，只不过不是标准库里的成员函数

唉，这里还存在一个更为基本的问题：我们不可能可移植地创建一个指向标准库里的成员函数的指针。

毕竟，如果想要创建一个指向函数的指针，不管该函数是不是成员函数，你都得知道该函数指针的类型，这就意味着你首先必须知道该函数的签名。 29

由于标准库里面的成员函数的签名无法确切得知，除非你去查看标准库实现的相应头文件，看看所需要的成员函数是否被添加了某些“皮卡布”形参，而且即便这次知道了，库的下一个发布版仍然有可能改动，总之，就是无法既可靠地使用指向标准库中成员函数的指针又得到可移植的代码。

小结

这确实让人有几分意外，我们可以将一个标准库设施（即 mem_fun）可移植地运用到几乎任何成员函数身上，可惜就是没法可移植地用到标准库中的成员函数上。同样令人感到意外的是，你可以将一个语言特性（即指向成员函数的指针）可移植地运用到任何地方，然而该语言自身的标准库除外。

通常标准成员函数的实现自由度对用户来说是不可见的，如果所要做的仅仅是调用这些函数的话，你永远也不会注意到有什么差别。然而，如果要使用指向成员函数的指针，或者约束器设施的话，你就得记住，将它们用在标准库本身的成员函数上是不可靠的。即便今天碰巧在你的特定标准库实现中某些特定的成员函数上能够工作了，等到库的版本一更新，说不定就又不能工作了。 30

第5条 泛型性的风味之一：基础

难度：4

在第6条的深入探讨之前，我们先来做一些准备工作，考虑下面这个简单的例子，它展示了C++中灵活的泛型代码。本条以及下一条中的代码示例都是从*Exceptional C++*[Sutter00，第42页]一书中摘选出来的。

JG问题

1. “C++模板提供了编译期多态的能力”，请解释这句话。

Guru问题

2. 下面这个函数的语义是什么？请尽量说全面一些，并务必解释为什么这里有两个模板参数而不是一个。

```
template <class T1, class T2>
void construct(T1* p, const T2& value) {
  new (p) T1(value);
}
```

解决方案

1. “C++模板提供了编译期多态的能力”，请解释这句话。

当我们在一个面向对象的世界当中考虑多态的时候，所考虑的是运行时多态，这种能力来自虚函数。基类建立起一个接口“合约”，该“合约”由一系列虚函数构成，派生类可以从基类进行派生，并在不违反合约所蕴涵的语义的前提下重写基类的虚函数。这么一来，那些期望使用基类对象（通过指针或引用来持有基类对象）来工作的代码使用派生类同样能够运行良好。

```
// 示例5-1(a): 我们司空见惯的运行时多态
//
class Base {
public:
  virtual void f();
  virtual void g();
};
class Derived : public Base {
  // 如果有必要的话，重写f和/或g
};

void h(Base& b) {
```

31

```
  b.f();
  b.g();
}

int main() {
  Derived d;
  h(d);
}
```

这是极其有用的特性，带来了若干运行时的灵活性。不过，运行时多态有两大主要缺点：首先，这些类型必须位于同一个继承自某个公共基类的类层次结构当中；其次，当虚函数在一个密集的循环当中被调用的时候，你可能会注意到它会带来一些运行时开销，因为通常每次对虚函数的调用都要通过一个额外的间接层——虚函数指针，同时，编译器负责根据你的意思将虚函数调用分发到派生类当中对应的函数。

而如果你在编译期就知道正在使用的是什么类型的话，就可以避免这两个缺陷：你可以使用那些并非具有继承关系的类型，只要它们支持你要的操作即可。

```
// 示例5-1(b): 新型的编译期多态。这是个强大的特性。
// 至今我们仍在不断探寻这种能力可能带来哪些令人惊奇的东西。
//

class Xyzzy {
public:
  void f(bool someParm = true);
  void g();
  void GoToGazebo();

  // ……更多函数……

};

class Murgatroyd {
public:
  void f();
  void g(double two = 6.28, double e = 2.71828);
  int HeavensTo(const Z&) const;

  // ……更多函数……

};
template<class T>
void h(T& t) {
  t.f();
  t.g();
}

int main() {
```

```
  Xyzzy x;
  Murgatroyd m;

  h(x);
  h(m);
}
```

只要 x 跟 m 这两个对象都提供了名为 f 和 g 的无参成员函数，h()就能够工作。在示例 5-1(b)中，实际上 Xyzzy 和 Murgatroyd 中的成员函数 f 和 g 分别具有不同的函数签名，而且它们各自还提供了其他的成员函数，不过 h()并不关心这些。只要的确可以无参地调用 f 跟 g，编译器就允许 h()进行这种调用。当然，当 f 和 g 被 h()调用时，它们也应当做一些对 h()来说有意义的事情！

也就是说，模板提供了强大的编译期多态能力。虽说在大多数编译器上，使用模板不当可能会导致难以读懂的错误信息，然而不能否认，模板同样也是 C++中最强大的特性之一。

2. 下面这个函数的语义是什么？请尽量说全面一些，并务必解释一下为什么这里有两个模板参数而不是一个。

```
// 示例5-2(a): construct。
//
template <class T1, class T2>
void construct(T1* p, const T2& value) {
  new (p) T1(value);
}
```

construct()函数模板负责在给定的内存位置构造对象，并同时以某个初始值来初始化它。这里使用 new 操作符的形式称为定位 new（placement new），这种形式的 new 并不为新对象分配内存，而只不过是将新对象放置到 p 所指的内存位置。任何以这种方式 new 出来的对象都应当通过显式调用其析构函数来销毁（正如第 6 条中的问题 1 所示），而不是使用 delete 表达式。

那么，为什么 construct()具有两个模板参数呢？难道使用一个模板参数我们就无法复制 value 对象了吗？如果 construct()只有一个模板参数的话，如果要从一个不同类型的对象进行复制，就得明确地给出参数类型：

33

```
// 示例5-2(b): 一个功能稍差的construct()，以及为什么说它功能稍差
//
template <class T1>
void construct(T1* p, const T1& value) {
  new (p) T1(value);
}

// 假设p1和p2皆指向未初始化内存（raw memory）。
//
void f(double* p1, Base* p2) {
  Base b;
  Derived d;
```

```
  construct(p1, 2.718);          // ok
  construct(p2, b);              // ok

  construct(p1, 42);             // 错误：T1究竟是double还是int呢？
  construct<double>(p1, 42);     // ok

  construct(p2, d);              // 错误：T1究竟是Base还是Derived呢？
  construct<Base>(p2, d);        // ok
}
```

这里有两处地方之所以会发生二义性是因为编译器没有足够的信息推导出模板形参 T1 的值，因而程序员不得不手动显式地给出一个模板实参。然而我们是否应当允许程序员一声不响地从一个 int 值构造出一个 double 值呢？或许吧，可能发生的最糟糕的事情就是损失一些精度。我们是否应该允许程序员一声不响地从 Derived 对象构造出一个 Base 对象呢？或许吧，如果 Base 允许这么做的话，对象切片就有可能发生，不过这也可以是合法的选择。

假如我们想允许程序员在不用显式给出类型名的情况下就能够完成上面这些事情，就需要使用原先给出的具有两个独立的模板参数的 construct()了。 34

第6条 泛型性的风味之二：够“泛”了吗

难度：7

评价一个泛型函数到底有多“泛”。答案不仅取决于其自身的实现，同样取决于其使用接口。不过，一个经过完美泛化的接口却有可能在一个简单（而且诊断起来很费事）的编程失误面前栽跟头。

Guru问题

1. 在如下的函数中存在着一个不易觉察的陷阱，是关于代码泛型性的。找出它，并给出修正它的最佳办法。

```
template <class T>
void destroy(T* p) {
  p->~T();
}

template <class FwdIter>
void destroy(FwdIter first, FwdIter last) {
  while(first != last) {
    destroy(first);
    ++first;
  }
}
```

2. 下面这个函数的语义是什么？其中模板形参 T 的要求又是什么？有可能将这些要求当中的任何一条消掉吗？如果可能的话，请说明如何做，并讨论这么做是好还是坏，给出相应的理由。

```
template <class T>
void swap(T& a, T& b) {
  T temp(a); a = b; b = temp;
}
```

35

解决方案

1. 在如下的函数中存在着一个不易觉察的陷阱，是关于代码泛型性的。找出它，并给出修正它的最佳办法。

```
// 示例6-1: destroy
//
template <class T>
void destroy(T* p) {
  p->~T();
```

```
}

template <class FwdIter>
void destroy(FwdIter first, FwdIter last) {
  while(first != last) {
    destroy(first);
    ++first;
  }
}
```

destroy()负责销毁一个对象或者一个区间内的所有对象。其第一个重载版本接受单个指针，并调用它所指对象的析构函数。第二个版本则接受一个以一对迭代器来表示的区间，并逐一销毁该区间当中的所有对象。

不过这儿存在着一个不易觉察的陷阱，即两个参数的 destroy(FwdIter,FwdIter)是被模板化了的，因而可以接受任何泛型迭代器，它会依次将区间当中的每个迭代器传递给单参的 destroy(T*)，而后者要求 FwdIter 必须为纯粹的指针！这就毫无必要地丧失了一些原本对 FwdIter 进行模板化而带来的泛型性。

准则：记住，（指向数组内元素的）指针永远都是迭代器，而迭代器并非总是指针。

这同样意味着，某些代码如果试图调用 destroy(FwdIter,FwdIter)，而给出的迭代器又不是指针类型的话，则可能会导致相当晦涩的错误信息，因为至少其中的一处错误是在 destroy(FwdIter,FwdIter)中调用 destroy(first)之处，这时通常情况下编译器会生成类似下面这样的有用的错误信息（从某个流行编译器给出的错误信息当中摘选出来的）：

```
'void __cdecl destroy(template-parameter-1,template-parameter-1)' : expects 2
arguments - 1 provided
'void __cdecl destroy(template-parameter-1 *)' : could not deduce template
argument for 'template-parameter-1 *' from '[the iterator type I was using]'
```

36

就我所见识过的错误信息之中，上面这些错误信息并非最晦涩难懂。只需稍微多花一点工夫去阅读，你就会发现它们其实（差不多）告诉了你问题所在。第一则消息告诉你编译器正试图对 destroy(first)进行重载决议，编译器试图将它跟 destroy()的第一个版本，即接受两个参数的版本进行匹配，从而导致了这行错误信息。而第二则信息则表示编译器试图将该调用匹配到 destroy()的第二个版本，即单参的版本。然而，编译器的两次尝试均告失败，分别是由于不同的原因：双参的 destroy()固然可以接受迭代器为参数，但却需要一对迭代器；而单参的 destroy()虽然只接受一个参数，却要求该参数的类型为指针。没辙！

话虽如此，在实际中，由于使用该函数的目的，以及考虑它会将对象“打回”未初始化内存这一点，我们几乎从来也不会想要将它用到除了指针之外的其他地方。虽说如此，我们也只不过需要对代码动一点小小的手脚就可以让 destroy()能够接受任意类型，而不仅仅是指针类型的迭代器。既然如此，何乐而不为呢？具体做法是在两个参数的 destroy()版本中将下面这行代码稍作

改动。

```
destroy(first);
```

改为：

```
destroy(&*first);
```

这种做法几乎总是行之有效的。我们先将迭代器解引用，获得容器内对象的直接引用，然后获取它的地址，这就确保了我们能够得到想要的指针。更详细地说就是，所有符合标准的迭代器都要求提供 operator*()，而且该操作符必须返回一个真正的 T&。这便是 C++标准之所以不支持代理容器的原因。如果你还想知道关于这方面更多的信息以及相关的议题，可参考 *More Exceptional C++*[Sutt02]中的条款 39 中关于表达式&*t.begin()的讨论（导致 destroy(&*first)不能正常工作的情况还是有的，虽然极其罕见，正如精明的读者 Bill Wade 所指出的，如果 T 重载了 operator&()并用它来返回其他东西而不是当前对象的地址，那么改进后的形式仍然是无法正常工作的，不过那是病态的做法，我从来也没有见过哪个设计有正当理由这么做）。

那么，这个故事的精神是什么呢？在使用一个泛型函数来实现另一个泛型函数的时候，须留神一些不易觉察的泛型性丧失。本例中就是一个不易觉察但非常严重的泛型性丧失，即两个参数的 destroy()在可接受的迭代器类型方面并不像代码作者原先设想的那样宽泛。甚至就连改进后的版本仍旧存在着一个更微妙的次要缺陷，即将 destroy(first);改为 destroy(&*first);之后，我们实际对 T 增加了一个额外的要求，即 T 上的 operator&()应当具有常规语义，换句话说，应当返回目标对象的地址。而如果我们不使用一个函数来实现另一函数的话，这两个陷阱则都可以优雅地回避掉。

37

注意，我并不是叫你不要使用模板来实现模板，我只是建议你在这么做的时候要留意它们之间潜在的相互影响。勿庸置疑，人们常常会使用模板来实现模板，而且常常能妥善地完成任务。例如，当程序员知道他们的某两个类型可以以一种更为高效的方式来互换值，他们通常就会将 std::swap()针对这两个类型特化，这么一来，以后如果你要写一个 sort()模板的话，sort()就应当使用 swap()来进行值交换，否则就无法针对具体的元素类型选出最高效的值交换途径。

2. 下面这个函数的语义是什么？其中模板形参 T 的要求又是什么？可能将这些要求当中的任何一条消掉吗？如果可能的话，请说明如何做，并讨论这么做是好还是坏，给出相应的理由。

```
// 示例6-2(a): swap。
//
template <class T>
void swap(T& a, T& b) {
 T temp(a); a = b; b = temp;
}
```

swap()只不过是通过复制构造函数和复制赋值操作符来将两个值进行交换而已，因而它要求 T 具有一个复制构造函数和一个复制赋值操作符。

如果你的答案就是这些的话，只能得一半分。在考察任何函数的语义的时候，最值得注意的

一个方面就是异常安全性，包括它提供了什么程度的异常安全保证等。本例中，swap()根本就不是异常安全的，例如我们可以假设 T 的复制赋值操作符会抛出异常，特别是如果 T::operator=()会抛出异常但它同时又具有原子语义（要么全部成功要么全部失败），再假设第二次复制赋值失败的话，我们就会因异常的抛出而离开 swap()函数，然而届时 a 的值已经被修改了[1]。更糟的情况是 T::operator=()不仅会抛出异常，而且不具有原子语义，那么就可能发生这样的结果，swap()异常退出，然而它的两个参数都被修改了，其中一个被修改后的值可能既不是 a 也不是 b 的值。这样的话，swap()的文档就必须像下面这样来描述。

- 如果 T::operator=()不抛出异常，那么除了 T 操作[2]的副作用之外，swap()就能够给出“要么全部成功，要么全部失败”的原子语义保证（参考[Sutter99]）。
- 否则，如果 T::operator=可能抛出异常的话：
  - 如果 T::operator=()具有原子语义，而且 swap()是异常退出，那么第一个实参就有可能被也有可能不被修改。
  - 否则，如果 T::operator=()不具有原子语义，而且 swap()是异常退出，那么两个实参皆有可能被也有可能不被修改，而且其中一个实参的值有可能既不是实参一的值也不是实参二的值。

38

有两个办法可以消除“T 必须具有一个复制赋值操作符”这一限制，第一个办法还能额外提供更好的异常安全性。

1. 第一个办法即特化或重载 swap()，假设我们有一个类 MyClass，该类遵循了一个常用手法，即提供一个不抛出异常的 Swap()成员函数。这么一来我们就可以针对 MyClass 来特化标准的 swap()函数了，如下：

```
// 示例6-2(b): 特化swap()。
//
class MyClass {
public:
  void Swap(MyClass&) /* throw() */ ;
  // ......
};

namespace std {
  template<> swap<MyClass>(MyClass& a, MyClass& b) { // throw()
    a.Swap(b);
  }
}
```

针对 MyClass 来重载标准的 swap()函数也可以达到同样的效果，如下：

1. 就好像剃头只剃了一半一样，换句话说，swap()整体上却并不具有原子语义。——译者注
2. 即 T 的复制构造函数。——译者注

```
// 示例6-2(c): 重载swap()。
//
class MyClass {
public:
  void Swap(MyClass&) /* throw() */ ;
  // ......
};

// 注意，这个重载并没有放在std名字空间当中。
swap(MyClass& a, MyClass& b) /* throw() */ {
  a.Swap(b);
}
```

即便 T 确实具有一个赋值操作符（从而原先那个 swap()也是可行的），这种做法通常也不失为一个好主意。

39

例如，标准库自身就针对 vector 重载[1]了 swap()，从而对 vector 调用 swap()实际上调用的是 vector::swap()。这一举措意义重大，因为 vector::swap()不用对 vector 内的元素做任何复制就可以实现交换，所以效率高得多。而这件事如果让示例 6-2(a)中的 swap()主模板来做，它就会首先创建其中一个 vector 的一份全新的副本（temp），然后将另一个 vector 复制到这个，再将 temp 复制到另一个 vector，这会导致一系列的 T 操作[2]，其时间复杂度为 O(N)，N 为互换的两个 vector 的大小的和。而针对 vector 特化过的 swap()则通常只需简单地对一些指针和整型变量进行赋值即可，耗常量时间（通常可以忽略不计）。一转眼的工夫就完成了。

这就是说，如果创建的新类型提供了类似于 swap 的操作，那么通常最好也为你的类特化一下 std::swap()（或者在其他名字空间中[3]提供自己的 swap()重载也行）。你的 swap()通常要比一般化的 std::swap()更高效，后者使的是蛮力，而你的 swap()则可以用巧劲，而且你的 swap()往往能够改善 swap()本身的异常安全性。

准则： 如果你的类型（跟只知道蛮干的std::swap()相比）有一个更好的途径可以用于交换其对象的值的话，请考虑为它特化std::swap()。

---

2. 消除“T 必须具有一个复制赋值操作符”这一限制的第二条途径是“先析构再重建”。其核心理念是通过 T 的复制构造函数而非复制赋值操作符来实现交换，当然这种做法的前提是 T 必须确实具有复制构造函数。

```
// 示例6-2(d): 不用复制赋值的swap()
//
template <class T>
```

1. 并非“特化”，因为无法偏特化函数模板。见第 7 条更多关于函数模板和特化的讨论。
2. 即复制构造。——译者注
3. 通常就是你的类定义所位于的名字空间，这样 Koenig 查找（类型依赖的名字查找）才会查找到它。

```
void swap(T& a, T& b) {
 if(&a != &b) {                          // 现在这个检查变得有必要了！
    T temp(a);

    destroy(&a);
    construct(&a, b);

    destroy(&b);
    construct(&b, temp);
 }
}
```

40

首先，如果 T 的复制构造函数可能会抛出异常的话，这种做法则肯定是不妥当的，因为这样一来你就会面临跟最初的 std::swap()一样的异常安全性问题。甚至只会更严重，你可能会遇到这样的情况：被操作的对象不仅具有不确定的值，甚至根本就不存在！

而如果我们知道 T 的拷贝构造函数一定不会抛出异常的话，该版本倒的确能够额外对付那些不能被赋值但可以被拷贝构造的类型，而且实际中也确实存在着许多这样的类型。但能够交换这些类型的值并不一定是什么好事，例如这种类型可能并不具有值语义，而且它可能具有 const 或引用成员，因而我们等于是在提供一种可以为类型强加值语义的机制，这可能具有误导效果，并导致意外或不正确的结果。

更糟的是，这种做法是在“玩弄”对象的生命期，而后者总是应该受到质疑的。这里我之所以说“玩弄”是因为它不仅改变了对象的值，而且还从真正意义上对被操作的对象的存在状态进行了改变。使用示例 6-2(d)中的 swap()的用户如果忘记它具有非常规的析构语义的话，则容易遇到意料之外的结果，

准则：如果必须操纵对象生命期，而且知道这么做是正确的，而且确信被操作的对象的拷贝构造函数永远不会抛出异常，而且你非常肯定当你的 swap()被运用到特定对象上的时候，这种非常规的“强加的”值语义不会带来什么问题的话，那么（只有在这些前提之下）或许才可以合理地决定针对某些特定情况（对象）使用这种做法。然而即便如此，也要记住不要将这种做法用在一个通用的，可能会意外地由任何类型来实例化的模板当中，而且一定得将它的不同寻常的行为记入文档，这样可怜的毫无戒备的隔壁程序员才能知道会发生什么样的行为，因为该技术已经完全属于 C++编码中的“非常规手段”了。

41

第7条 为什么不特化函数模板

难度：8

尽管本条的标题是个问句，但它同样也可以表达成陈述句：本条讲述的是“何时/为何不特化函数模板”这一主题。

JG问题

1. C++中主要有哪两种形式的模板，它们分别如何进行特化？

Guru问题

2. 在如下的代码中，最后一行代码调用的是f()的哪个版本？为什么？

```
template<class T>
void f(T);

template<>
void f<int*>(int*);

template<class T>
void f(T*);

// ……

int *p;
f(p);                    // 调用哪个f()?
```

解决方案

重要的区别：重载与特化

首先确保我们已对这两个概念了然于胸是很重要的，因此下面是一个快速的回顾。

1. C++中主要有哪两种形式的模板，它们分别如何进行特化？

C++中有类模板和函数模板之分。这两种模板的工作方式并不完全一样，最明显的区别在于重载，普通的C++类是不能重载的，因此类模板也不能够重载。另一方面，普通的C++函数如果名字相同（且函数签名不同）就会发生重载，因而函数模板也允许重载。这些规则相当合乎情理。我们目前所拥有的能力如示例7-1所示。

42

```
// 示例7-1: 类模板与函数模板，以及重载
//

// 一个类模板
```

```
template<class T> class X { /*...*/ };          // (a)

// 一个具有两个重载版本的函数模板
template<class T> void f(T);                    // (b)
template<class T> void f(int, T, double);       // (c)
```

这些非特化版的模板同样也被称为主模板。

此外，主模板还可以被特化。这一点使得类模板跟函数模板之间的差异性更大了，在本条款的后面部分，我们会看到这种差异性产生了重大的影响。类模板可以被偏特化或者全特化[1]。函数模板则只能够被全特化，不过，由于函数模板可以被重载，所以我们通过重载也能够达到类似偏特化的效果。下面的代码说明了这些差别：

```
// 示例7-1（续）：特化模板
//

// 将上例中的(a)针对指针类型做一个偏特化
template<class T> class X<T*> { /*...*/ };

// 将上例中的(a)针对int做一个偏特化
template<> class X<int> { /*...*/ };

// 一个独立的主模板，重载了(b)和(c)，注意，它并非(b)的偏特化，因为
// 函数模板并不允许偏特化
template<class T> void f(T*);                   // (d)

// (b)关于int的全特化
template<> void f<int>(int);                    // (e)

// 一个普通的函数，恰好重载了(b)、(c)和(d)，但并非(e)，后面我们将会
// 讨论这一点
void f(double);                                 // (f)
```

43

准则：记住，函数模板不能偏特化，只能重载。写一个看似函数模板偏特化的函数模板实际上是在写一个单独的主函数模板。

最后，让我们将注意力集中到函数模板上，考虑一下它们的重载规则，看看在不同的情况下到底哪个函数会被调用。规则相当简单，至少从总体上来看是这样，我们可以用经典的分类讨论法来描述该规则。

- 非模板函数是 C++中的一等公民。如果一个普通的非模板函数跟一个函数模板在重载解析的参数匹配中表现一样好的话，编译器会选择普通函数。
- 如果编译器发现没有合适的“一等公民”可选的话，那么主函数模板作为 C++中的二等公民就会被纳入考虑。具体选择哪个主函数模板则取决于哪个的参数类型匹配得最好，

1. 在标准的术语当中，全特化被称为“显式特化”。——译者注

如果这样还不能选出惟一的主函数模板的话，编译器则会根据下面一组相当晦涩的规则[1]确定哪个主函数模板是“最特化”的（重要提示：这里虽然称“特化”，然而实际上却跟“模板特化”没有任何关系，只是词汇上的不幸冲突而已）。

- 如果显而易见存在着一个“最特化”的主函数模板的话，该主函数模板就会被选中，如果这个被选中的主函数模板碰巧又针对所使用的模板实参（列表）做了特化的话，该特化版本就会被编译器选中，否则编译器将使用以正确类型实例化的主模板。
- 否则，如果有两个或多个“最特化”的主函数模板互相之间不能分出孰优孰劣的话，调用就是二义性的，因为编译器不能确定它们中哪个是更好的匹配。程序员只能以某种方式来对调用进行一些限定，明确指出想要调用的是哪个函数。
- 否则，如果没有任何主函数模板可以匹配调用的话，调用就是错误的，这时候程序员就有必要去纠正他们的代码。

这些规则加在一起作用的结果可以用下面这个例子来加以说明：

```
// 示例7-1（续）：重载决议
//
bool    b;
int     i;
double  d;

f(b);           // 调用(b)，T推导为bool
f(i, 42, d);    // 调用(c)，T推导为int
f(&i);          // 调用(d)，T推导为int
f(i);           // 调用(e)
f(d);           // 调用(f)
```

44

到目前为止，我都是在故意选择一些较为简单的例子，而下面将下潜到C++的深层去。

为什么不特化：Dimov/Abrahams的例子

考虑如下的代码：

```
// 示例7-2(a): 显式特化
//
template<class T>       // (a): 一个主模板
void f(T);

template<class T>       // (b): 一个主模板，重载了(a): 由于函数模板不能
void f(T*);             // 被偏特化，所以只能用重载来代替

template<>              // (c): (b)的一个显式特化
void f<int>(int*);
```

1. 即模板偏序规则[C++03:14.5.5.2]。——译者注

```
// ……

int *p;
f(p);                    // 调用(c)
```

示例 7-2(a)中最后一行的结果是意料之中的。然而，这里的问题是，为什么你会认为调用的是(c)？如果是歪打正着的话，当你看到下面的例子的时候可能就会大吃一惊了。毕竟，人们可能会说“那又如何，我针对 int 写了一个特化，那么显然这个特化就是应当被调用的那个了，”如果你是这么认为，可就大错特错了。

现在考虑问题 2 的代码，仍然是由 Peter Dimov 和 Dave Abrahams 给出的。

2. 在如下的代码中，最后一行代码调用的是 f()的哪个版本？为什么？

```
// 示例7-2(b): Dimov/Abrahams的例子
//
template<class T>
void f(T);

template<>
void f<int*>(int*);

template<class T>
void f(T*);

// ……

int *p;
f(p);                    // 这里调用的是哪个f()?
```

45

答案是……第三个 f()！下面重复一遍刚才的代码，不过这次加上了类似示例 7-2(a)中的注释，以便比较和对照这两个例子。

```
template<class T>        // (a): 跟前面一样的主模板
void f(T);

template<>               // (c): 显式特化，这次是对(a)进行特化
void f<int*>(int*);

template<class T>        // (b): 第二个主模板，重载了(a)
void f(T*);

// ……

int *p;
f(p);                    // 调用(b)！重载决议无视特化的存在，只在
                         // 主函数模板之间进行决议
```

如果这令你感到惊讶的话，你并不是惟一一个，当时它曾让许多专家大吃一惊。理解这个例子的关键其实很简单：模板特化并不参加重载。

只有主模板才会参加重载（当然，还有普通函数）。再次考虑我前面给出的关于重载决议规则的总结中某些比较突出的部分，这次我将一些特定的单词醒目地标了出来，如下：

- ……
- 如果编译器发现没有合适的一等公民可选的话，那么主函数模板作为 C++中的二等公民就会被纳入考虑。具体选择哪个主函数模板则取决于哪个的参数类型匹配得最好，如果这样还不能选出惟一的主函数模板的话，编译器则会根据下面一组相当晦涩的规则确定出哪个主函数模板是“最特化”的。
  - 如果显而易见存在着一个“最特化”的主函数模板的话，该主函数模板就会被选中，如果该被选中的主函数模板碰巧又针对被使用的模板实参（列表）作了特化的话，该特化版本就会被编译器选中，否则编译器将使用以正确类型实例化的主模板。
  - 等等

重载决议只会选出主模板（也可能选中普通函数，如果存在这么一个可用的普通函数的话）。只有在确定了到底调用哪个主模板之后，编译器才会着手四处寻找该主模板是否有某个合适的特化版本是可用的，如果找到了，那么就会采用那个特化版本。

准则：记住，函数模板特化并不参与重载决议。只有在某个主模板被重载决议选中的前提下，其特化版本才有可能被使用。而且，编译器在选择主模板的时候并不关心它是否有某个特化版本。

46

重要教训

如果你跟我一样，那么你第一次看到这些东西的时候或许也会问同样的问题：“呃，不过对我来说似乎是我特地写了一个针对 int*的版本，而且实参的类型又的确是 int*，这是个完全匹配，那么难道我的特化版本不应该被选用吗？”只可惜这是个错误的想法：如果想确保你的特化版本在类型完全匹配的情况下总是被选用，你就应该将它写成一个普通的函数，而不是模板特化。

至于模板特化为何不参与重载决议，只要稍加解释你就会发现其实理由很简单，因为引起人们惊讶的因素恰恰相反：标准委员会认为，如果仅仅由于你碰巧为某个特定的模板写了一个特化就导致编译器选择了不同的模板，人们就会感到惊讶。基于这个理念，又由于倘若有需要的话我们已经有方法可以确保我们的某个特殊版本得到采用（只需将它写成普通函数，而不是模板特化即可），就不难理解为什么模板特化不会影响重载决议。

准则：教训#1：如果你想要将一个主模板特化，同时又希望该特化版本能够参与重载决议（或者希望确保当它能够完全匹配用户调用的时候能被编译器选用）的话，只需将其写成普通函数即可。
推论：如果你确实提供了某个函数模板的重载，那么你应当避免为它提供特化。

然而，如果你是某个函数模板的作者而不是用户，又该怎么办呢？你是否能够做得更好一些，在第一时间为自己也为你的用户避免这类（以及其他）问题呢？实际上，答案是肯定的。

```
// 示例7-2(c): 阐释教训#2
//
template<class T>
struct FImpl;

template<class T>
void f(T t) { FImpl<T>::f(t); }        // 客户，请不要动这个函数!

template<class T>
struct FImpl {
  static void f(T t);                  // 客户，你可以对这个函数进行特化
};
```

47

准则： 教训#2：如果你正在写一个可能需要被特化的主函数模板的话，请尽量将它写成一个孤立的、永远不该被特化或重载的函数模板，并将其具体实现全部放入一个包含了一个静态函数的类模板当中。这么一来任何人都可以对后者进行特化（全特化或偏特化），而同时又不会影响到主函数模板的重载决议。

小结

对函数模板进行重载是允许的。编译器在进行重载决议时会平等对待所有的主模板，因此整个过程和结果跟你在普通的 C++函数重载中获得的经验应当完全吻合，即任何可见的模板都会被纳入到重载决议当中，然后编译器负责从中选出最匹配的。

而当你去特化函数模板的时候，事情就不那么符合直觉了。一方面，你不能对它们进行偏特化，而只能重载。另一方面，*函数模板的特化并不参与重载决议*。这意味着不管你写什么样的特化都不会影响重载决议的结果，这跟大多数人的直觉恰恰相反。不过，如果你写了一个跟模板特化具有同样签名的普通函数的话，编译器就会选择你的普通函数，因为在其他条件都不相上下的情况下，普通函数总是被认为是比模板更好的匹配。

倘若你要写一个函数模板的话，请尽量将它写成一个孤立的、永远不该被特化或重载的函数模板，并将它的具体实现全部放入一个包含了一个静态函数的类模板当中。这种众所周知的增加一个间接层的方法能够帮你避开函数模板的一些限制和黑暗角落。这样，使用你的模板的程序员就能够随心所欲地对包含了具体实现的类模板进行偏特化或显式特化，而再也不用担心会影响到函数模板的重载决议了。这不仅避开了函数模板不能偏特化这一限制，而且还免除了因函数模板特化不会参与重载而带来的一些出人意料的结果。问题解决了！

然而，如果你正在使用的是其他人写的“普通”函数模板（即不像我们刚才所述的使用类模板来实现的函数模板），而且你又想要加入一个自己的针对特殊情况进行处理的“特别版”并想让它参与重载，请千万不要将它写成函数模板特化，只需将它写成一个具有相应签名的普通函数（或主函数模板）即可。

48

第8条 友元模板

难度：4

如果你想要将一个函数模板的特化声明为友元的话，你应当怎么做？根据C++标准，你可以在两种合法的语法当中任选其一。然而，受到现实世界中编译器的约束，其中的一种语法普遍不支持，而另一种语法则在所有流行编译器的当前版本上是可行的……但是一个版本例外！

假设我们有一个函数模板，它会对其所操作的对象做一些“私事”。例如，考虑[Boost]中的boost::checked_delete()函数模板，它负责将接受的对象销毁，在它做的其他事中有一件事就是调用该对象的析构函数。

```
namespace boost {
 template<typename T> void checked_delete(T* x) {

  //……其他内容……

  delete x;
 }
}
```

现在，假设你想要将这个函数模板用到一个类上，而这个类的析构函数又碰巧是私有的。

```
class Test {
 ~Test() { }                    // 私有！
};

Test* t = new Test;
boost::checked_delete(t);       // 错误：Test的析构函数是私有的，
                                // 因此checked_delete不能调用它。
```

解决方案很简单，只需将 checked_delete 声明为 Test 的友元即可（除此之外惟一的选择就是放弃 Test 的析构函数的私有性，即将它声明为公用的）。还有比这更容易的吗？

实际上，在标准 C++当中有两条合法而简单的途径可以达到这个目的。只不过那也得编译器答应才行……

JG问题

1. 有一种显而易见且符合标准的语法可以将 boost::checked_delete 声明为 Test 的友元，请说出这种语法。

Guru问题

49

2. 为什么说这种显而易见的语法在实际中是不可靠的？请给出更为可靠的替代方案。

解决方案

本条会让你了解到现实的残酷性。将另一个名字空间中的模板声明为友元其实是说起来容易（在标准中）做起来难（在现实世界中的那些没能很好地“领悟”标准的编译器面前）。

总之，我有一些好消息、一些坏消息，最后再加上一些好消息。

- 好消息是，有两种完全 OK 且遵从标准的途径可以用来实现我们的目的，而且其语法也自然而然平淡无奇。
- 坏消息是，这两种标准语法都不可能在所有当前的编译器上皆有效。即便是一些最强大、最符合标准的编译器也不能完全支持这两种理应可用的、合法的和符合标准的语法。
- 好消息是，这两种做法中的一个的确在我测试过的所有当前编译器上都是可行的，但 GCC 除外。

下面就让我们来考察一下。

最初的尝试

1. *有一种显而易见且符合标准的语法可以将 boost::checked_delete 声明为 Test 的友元，请说出这种语法。*

本条是因Stephan Born在新闻组上提出的一个问题而起的，Stephan想让boost::checked_delete()成为类的友元。然而他遇到了问题。当他试图写一个友元声明，将 boost::checked_delete()的一个特化版本声明为 Test 类的友元时，代码无法通过编译。

这是他当时写的代码。

```
// 示例8-1: 授予友元关系的一种做法
//
class Test {
  ~Test() { }
  friend void boost::checked_delete(Test* x);
};
```

可惜，上面的代码不光是在 Stephan 的编译器上通不过编译，实际上在很大一部分编译器上同样也都通不过。简而言之，示例 8-1 中的友元声明具有如下的性质。

- 根据标准，这种做法是合法的，然而它却依赖于语言中的一个冷僻特性。
- 它在当前的许多编译器，包括一些非常好的编译器上都通不过编译。 50
- 只需稍作修改就可以令它不再依赖于语言的冷僻特性，修改后的版本在除了 gcc 以外的所有其他编译器上都能够通过编译。

我打算深入解释 C++语言允许我们声明友元的四种方式。这四种方式都很简单。我也乐于展示现实中的编译器是怎么做的，最后还会给出一个准则，说明如何编写可移植性最佳的代码。

为什么说它虽然合法但是冷僻

2. 为什么说这种显而易见的语法在实际中是不可靠的？请给出更可靠的替代方案。

在声明友元时，你有四种选择（在[C++03]的 14.5.3 节有详细的列举）。我将它们归结如下。

当声明一个友元，同时又没有任何地方使用 template 关键字的时候：

（1）如果友元的名字看起来像是一个模板特化，即显式给出了模板实参的话（例如 Name<SomeType>）

则该友元就是指那个模板的特化体。

（2）否则如果该友元的名字由一个类或名字空间所限定（例如 Some::Name），并且这个类或名字空间当中的确包含一个名字与友元声明中的名字相匹配的非模板函数的话

则这个非模板函数就是所声明的友元。

（3）否则如果该友元的名字由一个类或名字空间所限定（例如 Some::Name），并且该类或名字空间当中的确包含一个名字与友元声明中的名字相匹配的函数模板的话（推导出相应的模板参数）

则该友元就是指那个函数模板的特化体。

（4）否则友元声明中的名字肯定是非限定名，声明（或重复声明）了一个普通（非模板）函数。

显然，第（2）跟第（4）种情形只匹配非模板函数，因此要想将模板特化声明为类的友元，我们有两个选择，一是制造情形#1，二是制造情形#3。在我们的例子当中，有如下选择。

```
// 原始代码，合法，因为它属于情形#3
friend void boost::checked_delete(Test* x);
```

或者

```
// 添加"<Test>"，仍然合法，因为它属于情形#1
friend void boost::checked_delete<Test>(Test* x);
```

51

第一种做法是第二种做法的简略形式，只不过第一种做法的生效有一个前提条件，即仅当友元名字是限定名（这里是 boost::），且在同一个域中找不到匹配的非模板函数的情况下它才表示友元模板。虽说这两种做法都是合法的，但第一种做法利用了友元声明规则的一个冷僻特性，这一特性会令许多人乃至时下的大多数编译器都感到惊讶。我能给出不下三个理由来避免这种做法，虽然它从技术角度来说是合法的。

问题1：它并非总是行之有效的

正如我们已经注意到的，情形#3的语法是显式给出模板实参的做法的一个简略形式，然而它只有在友元名是限定名，且限定的类或名字空间当中不包含匹配的非模板函数的情况下才是行之

有效的。

特别地，如果限定的名字空间中具有（或者后来拥有了）一个匹配的非模板函数，那么编译器就会选择那个非模板的函数，这是因为非模板函数的存在意味着我们会先落入情形#2，而并非情形#3。有点微妙和惊讶，是吗？有点容易出错，是吗？让我们避开这种微妙的情况。

问题2：它让程序员感到惊讶

情形#3是脆弱的，当程序员看到这样的代码并试图弄明白它到底做了什么的时候，结果往往令人大吃一惊。例如，考虑下面这个稍作改动后的例子，我所做的仅仅是将boost::限定去掉了而已。

```
// 稍加改动后的例子，将友元的名字改成非限定名，结果就完全是两码事了
//
class Test {
  ~Test() { }
  friend void checked_delete(Test* x);
};
```

如果将boost::省略的话（即如果改成非限定名），你就落入了一个完全不同的情形，即情形#4中，而情形#4根本不会匹配出一个函数模板。想想看，我们仅仅省略了一个名字空间前缀就完全改变了友元声明的含义，我打赌绝大多数人都会同意这的确令人很意外。因此最好别去碰这个烫手山芋。

问题3：它令编译器感到“惊讶”

情形#3是脆弱的，而且令一些编译器也感到“惊讶”，因此即便我们不去理会前面的两个缺点，仅此一点就会导致它无法在实践中的使用。 52

现在让我们在一组广泛的当前编译器上测试情形#1跟情形#3，看看编译器们怎么说。编译器能够像我们一样理解标准吗（当然，是指我们在读了本条款之后的理解）？是不是所有强大的编译器给出的答复都像我们所预期的那样呢？两个问题的答案皆是否定的。

首先让我们来测试情形#3：

```
// 示例8-1，再来一次
//
namespace boost {
  template<typename T> void checked_delete(T* x) {

    // ……其他内容……

    delete x;
  }
}

class Test {
```

```
  ~Test() { }
  friend void boost::checked_delete(Test* x);    // 原始代码
};

int main() {
  boost::checked_delete(new Test);
}
```

在你的编译器上测试这些代码，然后将结果跟我们的比较。如果你曾在电视上看过家庭之争节目的话，现在就可以想像 Richard Dawson 在说："调查表……明……"（见表 8-1）。

表 8-1　在不同编译器上编译示例 8-1 的结果

编译器	结果	错误信息
Borland 5.5	通过	
Comeau 4.3.0.1	通过	
Digital Mars 8.38	错误	Symbol Undefined?checked_delete@@YAXPAV-Test@@@Z(void cdecl checked_delete(Test *))
EDG 3.0.1	通过	
Intel 6.0.1	通过	
gcc 2.95.3	错误	'boost::checked_delete(Test *)' should have been declared inside 'boost'
gcc 3.4	错误	'void boost::checked_delete(Test*)' should have been declared inside 'boost'
Metrowerks 8.2	错误	friend void boost::checked_delete(Test* x); name has not been declared in namespace/class
MS VC++ 6.0	错误	nonexistent function 'boost::checked_delete' specified as friend
MS VC++ 7.0(2002)	通过	
MS VC++7.1(2003)	错误	'boost::checked_delete' : not a function
MS VC++8.0(2005)beta	通过	

在本例中，调查表明实际上编译器并不能很好地识别这种语法，甚至同一个编译器的各个版本间也存在着分歧。当然，毫不奇怪地，Comeau、EDG 以及 Intel 编译器都接受该语法，因为它们都是基于 EDG C++语言实现而开发的。我一共测试了五种不同的 C++语言实现，其中就有三种（Digital Mars、gcc 和 Metrowerks）不接受这种做法，另外两个（Borland 和 EDG）则接受，而 Microsoft 的编译器的不同的版本则有不同的表现。

那么让我们再来测试另一个符合标准的语法，即情形#1。

```
// 示例8-2: 声明友元的另一种途径
//
namespace boost {
  template<typename T> void checked_delete(T* x) {
    // ……其他内容……
    delete x;
  }
}
```

```
class Test {
 ~Test() { }
 friend void boost::checked_delete<>(Test* x);  // 替代方案
};

int main() {
 boost::checked_delete(new Test);
}
```

或者等价地，我们也可以这么写：

```
friend void boost::checked_delete<Test>(Test* x);  // 等价的做法
```

我们的调查表明，编译器对以上两种语法的支持明显要好得多（见表 8-2）。

表 8-2 在不同编译器上编译示例 8-2 的结果

编译器	结果	错误信息
Borland 5.5	通过	
Comeau 4.3.0.1	通过	
Digital Mars 8.38	通过	
EDG 3.0.1	通过	
Intel 6.0.1	通过	
gcc 2.95.3	错误	'boost::checked_delete(Test *)' should have been declared inside 'boost'
gcc 3.1.1	错误	'void boost::checked_delete(Test*)' should have been declared inside 'boost'
gcc 3.4	错误	'void boost::checked_delete(Test*)' should have been declared inside 'boost'
Metrowerks 8.2	通过	
MS VC++ 6.0	错误	nonexistent function 'boost::checked_delete' specified as friend
MS VC++ 7.0(2002)	通过	
MS VC++7.1(2003)	通过	
MS VC++8.0(2005)beta	通过	

54

因此情形#1 当然让人感觉更为放心——示例 8-2 除了在 gcc 上通不过编译之外，在其他所有当前编译器上都畅通无阻，而在旧的编译器中则除了 VC6.0 之外其他都没问题。

题外话：难道说是名字空间困扰了编译器？

注意，如果要声明为友元的函数模板并不位于其他名字空间中，我们就可以在几乎所有的编译器上使用情形#1 的做法。

```
// 示例8-3: 假设checked_delete并不位于名字空间中……
//
template<typename T> void checked_delete(T* x) {  // 不再位于boost::中
 // ……其他内容……
 delete x;
}

class Test {
```

```
  friend void checked_delete<Test>(Test* x);    // 不再需要boost::前缀了
};

int main() {
  checked_delete(new Test);
}
```

调查表明（见表 8-3）：

表 8-3 在不同编译器上编译示例 8-3 的结果

编译器	结果	错误信息
Borland 5.5	通过	
Comeau 4.3.0.1	通过	
Digital Mars 8.38	通过	
EDG 3.0.1	通过	
Intel 6.0.1	通过	
gcc 2.95.3	通过	
gcc 3.4	通过	
Metrowerks 8.2	通过	
MS VC++ 6.0	错误	语法错误（无法处理这种语法）
MS VC++ 7.0(2002)	错误	错误地将友元声明解析成了声明一个全新的普通非模板函数（虽然这里使用了模板语法）
MS VC++7.1(2003)	通过	
MS VC++8.0(2005)beta	通过	

因此，大多数不能正确处理示例 8-1 的编译器的主要问题，都出在将一个位于另一名字空间中的函数模板的特化体声明为友元的时候。

55

两个不是办法的办法

当时这个问题在新闻组上贴出来之后，一些人的回复就建议写一个 using 声明（或者也可以用 using 指示），并将友元声明改成非限定名。

```
namespace boost {
  template<typename T> void checked_delete(T* x) {
    // ……其他内容……
    delete x;
  }
}

using boost::checked_delete;
// 或者 "using namespace boost;"

class Test {
~Test() { }
friend void checked_delete(Test* x);   // 并不是指模板特化
};
```

56

上面的友元声明属于情形#4："4. 否则该名字必须为非限定名，并且声明（或重新声明）了一个普通（非模板）函数。"因此这种做法实际上是在外围名字空间作用域中声明了一个全新的普通函数，叫作::checked_delete(Test*)。

如果你试着编译上面的代码，就会发现这些编译器中许多都会报错，说 checked_delete()还没有被定义。然后如果你试图利用这个友元关系，将一个私有成员调用放到 boost::checked_delete()模板中的话，会发现所有的编译器都开始"抱怨"了！

终于，一个专家给出了他的建议：只需对上例稍作改动，在使用 using 语句的同时也使用模板语法<>。

```
namespace boost {
 template<typename T> void checked_delete(T* x) {

  // ……其他内容……

  delete x;
 }
}

using boost::checked_delete;
// 或者"using namespace boost;"

class Test {
 ~Test() { }

friend void checked_delete<>(Test* x);    // 合法吗？

};
```

然而上面的代码也许并非合法的 C++代码，标准并没有明确指出这种做法到底合不合法，标准委员会的 C++ Standard Core Language Issue List 中就有这个问题，即像上面这样的代码究竟是否应该合法的问题，有些人的意见是它不应该是合法的，而且实际上几乎所有我所尝试过的当前编译器都拒绝它。

为什么人们会觉得这种做法是不合法的呢？问题在于一致性，因为 using 关键字的作用是让程序员更容易地使用名字，例如在调用函数的时候，或者在声明变量或参数时给出类型名的时候。而声明就完全是另外一回事了，因为你必须在主模板所在的名字空间中才能声明它的特化版本（不能通过 using 从而在另一个名字空间中声明其特化版本），因此在声明友元的时候你也应当恪守一致性，既然想将一个模板的特化体声明为友元，那么就应当指出它的主模板所在的名字空间（而不仅仅是通过一个 using）。

57

小结

要想将一个函数模板特化声明为友元，你可在以下两种方法中任选其一。

```
// 来自示例8-1
```

```
friend void boost::checked_delete (Test* x);

// 来自示例8-2: 加上"<>"或"<Test>"
friend void boost::checked_delete<>(Test* x);  // 或"<Test>"
```

本条说明在 C++中仅仅忽略小小的<>或<Test>（示例 8-2）就有可能损失相当多的可移植性。

准则：（在代码中）明确地表达出你的意图。如果你所指的确实是一个模板，然而你的表达方式可能存在歧义的话，可以在模板名字的后面加上模板实参列表（这个列表可能是空的，即<>）。

避开语言的冷僻特性，包括那些虽然可以说是合法但容易使程序员甚至编译器迷惑的地方。

当你想将一个函数模板声明为友元的时候，一定要显式地加上一个模板列表（<...>），哪怕是空的模板列表<>也行，因为它至少能够明确表示你所指的是一个模板。

```
namespace boost {
 template<typename T> void checked_delete(T* x);
}

class Test {
 friend void boost::checked_delete (Test* x);        // 糟糕的做法
 friend void boost::checked_delete <> (Test* x);     // 好的做法
};
```

然而，如果你的编译器目前连这两种合法的语法都不允许的话，你就只能将某些函数放到公用区段了[1]，不过同时还要记住加上一句注释，说明为什么这么做，并且做个记号，以便在将来升级了编译器之后及时改回去。

58

1. 当然，并非只有这一种弥补手段，只不过其他手段都显得太笨拙了。例如，可以在 boost 名字空间中为想声明为友元的函数模板创建一个代理类，然后将这个代理类声明为类的友元。

第 9 条　导出限制之一：基础

难度：7

人们认为export能做些什么，而实际上它能做什么，为什么说在C++标准中包含的主要语言特性里面，export是最普遍忽略的一个。

JG问题

1. “包含模式”对模板来说意味着什么？

Guru问题

2. “分离模式”对模板来说意味着什么？

3. 对于普通函数来说包含模式的主要缺陷有哪些？对于函数模板来说呢？

4. 对于普通函数来说，标准 C++分离模式对于问题 3 中的缺陷能有哪些帮助？而对于函数模板来说呢？

解决方案

标准 C++模板导出（export）特性是一个被普遍误解的特性，它的能力比大多数人最初认为的更有限，而它造成的影响却比大多数人最初认为的更大。本条以及下一条将对我们迄今为止在模板导出特性方面积累的经验做一个仔细的考察。

在写作本书的时候，有且仅有一个商业 C++编译器尚支持 export，即 Comeau[1]编译器，它是基于 Edison Design Group（简称 EDG[2]）的 C++语言实现前端开发的，后者是当时（以及到目前为止）惟一支持 export 特性的 C++实现，发布于 2002 年。而在现实项目中使用 export 的经验则仍然少之又少，尽管要是有更多 C++实现加入 export 支持的话情况或许会有所改观。然而话虽如此，仍然有些东西我们是知道的，另外，export 特性最初的实现者也获得了不少经验。 59

下面就是本条以及下一条将要讨论和涉及的内容。

- export 究竟是什么？其用途是什么？
- 人们广泛认为 export 应当解决的问题，以及为什么它实际上却不像大多数人所想的那样解决了那些问题。
- export 的当前形势，包括到目前为止我们在实现 export 特性的过程中得到的经验。
- export 怎样（通常是以一种不明显的方式）对 C++语言中其他看似与它无关的部分的基

1. 见 www.comeaucomputing.com。——译者注
2. 见 www.edg.com。——译者注

本意义产生影响。

- 一些关于如何有效地利用 export 特性的建议，当然，前提是如果你碰巧有一个支持 export 特性的编译器。

两种模式

C++标准支持两种不同的模板源代码组织方式，其中之一就是我们年复一年地使用着的包含模式，另一种相对较新鲜的模式是分离模式。

1. “包含模式”对模板来说意味着什么？

使用包含模式时，从源代码角度来看，模板代码和内联代码没什么区别（尽管模板并不一定要声明为内联的）：模板的全部代码对于使用它的客户代码来说必须是完全可见的。这被称为包含模式，因为基本上我们必须在模板头文件中包含所有的模板定义代码[1]。

如果你了解目前的 C++模板技术的话，就应当知道包含模式。包含模式是近十年来惟一得到实际运用的模板代码组织方式，因为到目前为止它仍然是惟一被标准 C++编译器（广泛）支持的模板代码组织方式。到目前为止你所看到的所有 C++书籍和文章中的模板代码无不是以这种方式来组织的。

60

2. “分离模式”对模板来说意味着什么？

另一方面，（模板的）分离模式以允许模板“分离式”编译为目的（这里“分离式”之所以加双引号是有原因的）。在分离模式下，模板定义对调用者不一定得是可见的。我很想在这句话后面加上“就像普通函数那样”，然而那样做是不尊重事实的。在我们的概念中似乎这两者并没有什么区别，然而它们实际表现出来的效果却是截然不同的，后面我们看到一些令人惊讶的部分时就会明白了。（模板的）分离模式是相对较新的模式，它是在 20 世纪 90 年代中期加入 C++标准的，然而其第一个商业实现却直到 2002 年才由 EDG 实现[2]。

下面我将用一整段的内容深入描述一些有关编译器实现的细节。首先我们得牢牢记住包含模式跟分离模式之间的一个细微但重要的区别，即其实它们的不同之处在于它们是不同的源代码组织方式。也就是说，它们的区别之处在于如何安排和组织源代码。它们并不会影响模板实例化的方式，换句话说，不管你使用包含模式还是分离模式，编译器在实例化模板的时候都是做完全一样的工作。记住这一点很重要，因为它一定程度上解释了为什么 export 的局限性（待会我们就会接触到）会令许多人惊讶，尤其是它还解释了为什么使用 export 不大可能将构建时间降低到普通函数的分离式编译通常达到的那种程度。例如，不管你使用哪种模式，编译器都仍然能够进行某

1. 另外还有一些等价的做法，例如将模板定义分离到一个单独的.cpp 文件中，然后让该模板的.h 头文件#include 这个.cpp 文件，这样做的效果也一样。
2. 注意，Cfront 在大约十年之前就有了一些类似的功能。然而 Cfront 的实现太慢了，况且它还是基于“大部分时候能工作”的启发式方法来工作的：当 Cfront 用户在构建过程中碰到一些与模板有关的问题时，Cfront 就会不管三七二十一先将模板实例化缓冲区全部清空，然后再重头开始实例化所有的东西。

些优化，如根据（而不是强制实施）惟一定义规则（ODR）对同样的一组模板实参只进行一次实例化，而不管这个实例在整个工程中有多么常用或者用得多么普遍。这种优化及实例化策略对于编译器作者来说总是可行的，而不管实际上源代码的物理组织方式是包含模式还是分离模式，尽管分离模式允许这种优化，但包含模式同样允许。

阐释问题

3. 对于普通函数来说包含模式的主要缺陷是什么？对于模板函数来说呢？

为了阐释这个问题，让我们先来看一些代码。

61

我们将会分别以包含模式和分离模式来组织一个函数模板的代码，出于比较的目的，我打算同样给出一个普通函数在内联与分离编译模式下的代码组织形式。这样做有助于凸显当前的普通函数分离编译模式与 export 所支持的模板“分离”编译模式之间的区别。这两者是不同的，虽然用于描述它们的术语是相同的。事实上，正因如此我才在后一个“分离”上加上双引号。

考虑如下的代码，它们是一个普通的内联函数，以及一个以包含模式组织的函数模板：

```
// 示例9-3(a): 一个我们司空见惯的内联函数
//

// --- file f.h，给用户使用的---
namespace MyLib {
 inline void f(int) {
 // 优雅而高超的实现，多年来工作的结晶
 // 使用了一些其他的辅助类和函数
  }
}
```

下面是一个函数模板在包含模式下的代码组织形式，跟上例恰巧形成对比：

```
// 示例9-3(b): 一个毫不知情的可怜的模板，使用了包含模式
//
// --- file g.h，给用户使用的 ---
namespace MyLib {
 template<typename T>
 void g(T&) {
  // 同样漂亮的实现，多年来工作的结晶
  // 使用了另一些辅助类和函数。模板函数不一定要声明为内联，
  // 不过函数体仍然需要全部放在这个头文件当中
 }
}
```

C++程序员应当熟悉这两段示例代码中存在的问题。

- 实现代码暴露：全世界都可以看到 f()和 g()的代码，而后者可能是专有代码。当然，这件事情本身或许并没那么糟糕，这个问题我们待会再说。

- 源代码依赖性：f()和 g()的所有调用方都依赖于它们所调用函数的函数体内的实现细节，因而每次这些函数的作者改过函数体之后，其所有调用方都必须重新编译。另外，如果 f()或 g()的定义中使用了其他类型，那么调用它们的客户代码同样也得依赖于这些类型的定义。

62

export的实际能力

我们有办法解决或者至少缓解这些问题吗？

4. 对于普通函数来说，标准 C++分离模式对于问题 3 中的缺陷能有哪些帮助？

对于普通函数来说，答案再简单不过了，我们只需改用分离编译模式即可。

```
// 示例9-4(a): 再普通不过的分离式编译（普通函数）
//
// — file f.h，给用户使用的 —
namespace MyLib {
  void f(int);                    // MYOB
}
// — file f.cpp，可以不发布给用户—
namespace MyLib {
  void f(int) {
    // 优雅而高超的实现，多年来工作的结晶
    // 使用了一些其他的辅助类和函数，不同的是现在这个函数体
    // 跟函数声明分开了
  }
}
```

毫不奇怪，该方案解决了我们上面所提到的“实现代码暴露”和“源代码依赖性”问题，至少对于 f()是这样（同样的手段也可以用在类的“分离式编译”上，即 Pimpl 惯用法，参见 *Exceptional C++* [Sutter00]）。

- 定义/实现代码不再暴露给外界：现在，如果愿意的话，我们当然仍可以发布实现的源代码，但假若不愿意的话，我们就完全不必发布它们了。注意，许多流行的库，即便是那些对专有权保护得非常严密的，有时也会因用户需要源代码改善程序调试或者其他原因而同时发布实现源代码（有可能需要用户付额外的差价）。
- 消除了源代码依赖性：调用方不再依赖于 f()的内部实现细节，因而每当其定义体改变，调用它的代码只需重新连接一下即可。这往往能够令构建速度快上一个数量级乃至更多。类似地，f()的调用方不再依赖于那些只是在 f()的定义体中被使用的类型，这也会缩短构建时间，只不过效果较小。

63

对于普通函数来说，所有这些听起来都挺不错，只不过对这些内容我们早就耳熟能详了。我们从 C 开始，乃至在 C 时代之前就开始这么干了（那是很久很久以前的事了）。

真正的问题是：对于模板来说，上面的这些讨论还成立吗？如果不成立，情况又是怎样的呢？

export 背后的理念是对模板也能够实现类似于普通函数分离式编译那样的效果。人们可能会天真地认为下面这样的代码能够像示例 9-4(a)那样具有类似的优势。然而如果你这么想的话可就错了，不过别气馁，这么想的人并不止你一个，许多人在这个问题上都栽过跟头，包括一些世界级的 C++专家。考虑如下的代码：

```
// 示例9-4(b): 一个依赖性“稍有减弱”的函数模板
//
// — file g.h，给用户使用的 —
namespace MyLib {
 export template<typename T>
 void g(T&);                    // MYOB
}
// — file g.cpp，??也给用户使用?? —
namespace MyLib {
 template<typename T>
 void g(T&) {
  // 优雅而炫目的实现，多年来努力的成果
  // 使用了一些其他类或模板，现在被“分离式”编译了！
 }
}
```

令许多人非常惊讶的是，对于 g()来说这种做法并不能像 f()那样解决那两个问题。它也许可以缓解其中一个问题。下面将逐一讨论这些问题。

问题一：源代码暴露

模板的分离式编译并不能解决这个问题，源代码暴露问题依然存在。

C++标准当中并没有说或者暗示你可以仅通过一个小小的 export 关键字就能避免发布 g()的完整源代码。实际上，就 export 现有的惟一实现而言，编译器仍然得提供被 export 之模板的完整定义，即全部源代码[1]。原因之一是 C++编译器在实例化这些模板的时候仍会需要用到它们的完整定义。我用一个例子来说明为什么是这样。C++标准中对实例化模板的时候会发生什么情况是这样来描述的：

64

> “[依赖]名字是‘非绑定’的，编译器对它们的名字查找会在模板实例化处，在模板定义的上下文跟实例化的上下文中进行。”——[C++03]的 14.6.2 节

1. 通常人们会问：“但难道我们不能发布加密后的源代码吗？”答案是任何仅由程序解密而无需用户介入（例如每次键入密码）的加密手段都是不牢靠的。同样，以前也有过一些公司出于各种各样的目的，如为了保护包含模式的 C++模板代码，而尝试过“加密”或者混淆源代码这些手段，但最终这些努力基本都放弃了，因为这些做法在实际当中会给用户带来麻烦，而且也并不真的能很好地保护源代码，更何况源代码原本就几乎不需要这种保护，因为还有其他更好的方法来保护知识产权。

依赖名字是指那些依赖于某个模板形参的名字，大多数有用的模板都包含依赖名字。在模板的实例化/使用处，编译器会在两个地方查找依赖名字。一是实例化处的上下文，这很简单，因为编译器正处于这个点上。然而还有另一个上下文，即模板定义的上下文，而这正是问题的关键，因为这便意味着编译器不仅需要知道模板的完整定义，而且还需知道包括该定义在内的其所在文件的上下文，包括哪些相关的函数签名位于作用域内等等，这样重载决议以及其他工作才能顺利进行。

从编译器的角度来看示例 9-4(b)，情况是这样的：你的库里面有一个导出的函数模板 g()，其定义体被很好地安置在了其头文件之外。然后你将库发布了。一年之后，一个阳光明媚的下午，某个用户在他的 h.cpp 编译单元中使用了你的模板，他决定用早晨刚定义出来的新类 CustType 来实例化你的模板，即 g<CustType>……，那么，这种情况下编译器要生成目标代码应该做些什么呢？它应该在实现文件中四处寻找 g()的定义。而这意味着什么呢？这意味着 export 并没有消除这种对于模板定义的依赖性，而只不过是将它隐藏了而已！

导出的模板并非是我们通常（对于普通函数）所说的真正的“分离式编译”。导出模板通常不能在使用之前就分离编译成目标代码。一方面，这是因为编译器直到使用模板的确切地点才能知道用哪些模板实参来实例化它。因此导出模板至多只能算是“分离式部分编译”或者“分离式解析”。而实际上，在每次实例化时模板的定义都要使用对应的一组模板实参编译一次。（这里有一点跟 Java 和.NET 库是类似的，后者的字节码或 IL 可以被逆向工程，从而得到非常类似于源代码的东西。）

65

准则：记住，export并不能带来像普通函数那样真正的分离式编译。

问题二：依赖性与构建时间

第二个问题同样也没有得到解决：依赖性被隐藏了起来，但依然存在。

每次模板的定义体改变时，编译器都要将使用它的客户代码统统重新实例化。而在这个过程中，使用了 g()的编译单元仍然跟 g()的内部实现密不可分，后者包括 g()的定义体以及仅在 g()的定义体中使用的类型。

而当日后被客户使用，且实例化上下文皆已知的情况下，模板的（实现）代码仍然需要全部经过编译器编译。下面就是我们必须记住的关键概念。

准则：记住，export只能隐藏依赖性，并不能消除依赖性。

的确，使用了 export 之后，调用方不再显式地依赖于 g()的内部实现细节了，但也只不过是 g()的定义不再被公然地通过#include 带入调用方的编译单元中而已，我们可以说依赖性在“用户阅读源代码”层面上被隐藏起来了。

然而这还不是故事的全部，因为这里我们关心的是“编译器必须进行的编译”依赖性，而不是“程序员一边喝着咖啡一边读代码”时的依赖性，而模板定义的编译依赖性仍然存在。诚然，当模板定义改变时，编译器或许并没有必要将每个使用了该模板的编译单元全部重编译一遍，然而它必须至少重编译足够多的编译单元，以便使得所有实例化并使用过的模板实例都已“刷新”。这种情况下编译器并不能像在真正的分离式编译模式下那样仅仅重新连接目标代码即可。

注意，即便是在包含模式下，例如代码是像示例 9-4(b)中那样组织的，只不过将 export 关键字移除，同时加上一行#include "g.cpp"到 g.h 中，编译器也可以“聪明”到像上面所描述的那样来处理重编译事宜，即仅仅重编译足够多的编译单元从而令所有实例化并使用过的模板实例都得到“刷新”即可。编译器之所以可以进行这种优化是仰赖于（而不是强制实施）惟一定义规则的，例如，根据惟一定义规则，编译器可以合理假设如果出现两个或两个以上的模板实例对应于同一组模板实参的话，那么这些实例肯定是相同的，这样一来编译器就只需实例化其中的一处，而无需一一实例化并检查它们是否真的一致。

此外，还值得注意的一点是，许多模板都会使用其他模板，因而编译器就会重编译那些已用的模板（以及它们的编译单元），依此类推，后者用的模板也会被重编译，如此递归，直到不能再继续为止。（如果听到这里你挺庆幸自己不用去实现 export 特性的话，那纯属正常反应:-)） 66

即便有了 export 特性，也并非就意味着那些使用了 export 模板的客户代码在前者被改动后“只需要重新连接（relink）即可”。跟普通函数的真正的分离式编译不同，到写作本书为止关于 export 特性通常会导致构建过程更快、更慢还是无甚改观这个问题仍没有明确的答案。

小结

本章我们考察了 export 特性背后的动机，以及为什么对于模板来说它并非是像普通函数那样的真正的分离式编译。许多人认为 export 即意味着可以不用发布模板库的实现源代码，或者构建速度会变得更快。然而实际上 export 根本没有保证会带来这两个优点。到目前为止 C++社区的经验是：即便你使用了 export，模板的源代码或者与它等价的东西仍然必须随库一同提供，同时构建速度可能并无甚改观，甚至更慢，鲜有更快的情况，从原则上来说这是因为依赖性虽然被掩盖起来了但依然不可否认地存在着，而只要存在依赖性，编译器就仍然需要像往常那样做一系列的工作（或许更多）。

下一条将会讨论为什么 export 致使 C++语言变得更复杂，并让它变得更难于使用，我们将会看到，export 实际上改变了语言的其他某些部分的基本意义，而且人们当时并没有很清晰地认识到这一点。我们同样还会看到当初的一些关于如何有效利用 export 的建议，如果你碰巧有一个支持 export 特性的编译器的话。 67

第 10 条 导出限制之二：相互影响，可用性问题以及准则

难度：9

export与现有的C++语言特性之间是如何相互影响的？安全使用它的首要准则是什么？

JG问题

1. export 的当前形式是从何时起在 C++标准中确立的？其首次实现是在何时？

Guru问题

2. export 以何种方式改变了 C++语言其他特性的基本意义？简要地解释它们之间的相互影响。

3. export 对程序员有所影响？

4. export 究竟有何实际的潜在优点？

解决方案

这是该主题的第二部分。在上一部分，我们讨论了如下几个方面：

- 什么是 export，该特性的使用意图是什么？我们分析了“包含”和“导出”这两种模板源代码组织模型之间的异同，以及为什么说它们不同于普通函数的内联和分离式编译之间的差别。
- 人们普遍认为 export 应当解决的问题，以及为什么实际上它并没有众望所归地解决这些问题。

尽管有着很多的期望，然而 export 并不能够为模板带来与非模板的分离式编译一样的真正的分离式编译。许多人都期望，export 意味着模板库可以不必发布全部源代码定义（或者其等价物），同时构建速度也会有所提高。然而实际上 export 没有提供这两个保证。

68

到目前为止，C++界所获得的关于 export 的经验表明，使用 export 仍需发布全部源代码（或者其等价物），而且人们尚不知道在实际使用当中 export 究竟会使构建速度变得更快、更慢，还是相差无几。为什么？主要是因为依赖性仍然存在，只不过被隐藏了起来，而且在一般应用中编译器仍至少需要做等量的工作。简而言之，虽然人们很自然会认为 export 能够为模板带来真正的分离式编译，赋予开发者只需发布模板头文件与目标代码文件的能力，然而这是个误解。更进一步说，被 export 的东西在某种意义上类似于 Java 和.NET 的库，Java 的字节码与.NET 的 IL 代码都可以通过逆向工程得到非常类似于源文件的内容，所以说被 export 的东西并非传统意义上的目标

（二进制）代码。

本条就以下主题作一些讨论：

- export 当前的形势，包括目前在实现 export 的过程当中积累起来的经验。
- export 会以某些（通常并不明显的）方式改变 C++语言当中看上去与它无关的某些部分的基本意义。
- 当碰巧使用的是支持 export 的编译器时，一些关于有效使用 export 的建议。

不过首先还是来回顾一些有趣的历史。

历史回顾：1988-1996

1. export 的当前形式是从何时起在 C++标准中确立的？其首次实现是在何时？

答案分别是 1996 年和 2002 年（你也许觉得某个特性的第一次实现通常应该在它被标准化之前，这情有可原，C++标准在对待 export 的时候采取了“先标准化再实现”的反常态做法，但这并不是惟一的例外）。

正因如此，再加上对于 export 存在一些中肯的批评，似乎有理由能对提出这个馊主意的家伙群起而攻之。但这是不公正的，也缺乏君子之风。本条之所以有这些背景介绍就是为了持论公允，因为在 export 这个问题上，人们很容易走向极端，旗帜鲜明地赞成或反对。

倘若 export 不能提供众所期望的好处，那为什么它还会存在于语言之中呢？原因非常简单：在 20 世纪 90 年代中期，C++标准委员会成员普遍认为颁布一个不具备模板的分离式编译特性（C 已经为普通函数做到了这一点）的标准是不完备且令人尴尬的。简单的说，根据一贯的原则，export 就被留在了标准草案之中。

69

原则往往是好事情。原则的力量永远不该被小觑，尤其是对我们这些事后诸葛亮而言，因为我们是在事情发生之后若干年才来看待这件事情，而这时候关于这件事情人们已经有了多年的认识了。（当然，我说的“我们这些事后诸葛亮”，也包括我自己在内。虽说事隔多年以后，我已经担任 ISO C++标准委员会的主席，但在这之前我并没有参加委员会的会议。）

1995 年和 1996 年的时候，模板本身还算是相当新颖的技术：

- C++模板的第一份设计方案是 Bjarne Stroustrup 在 1988 年 10 月提出的[Stroustrup88]。
- 到了 1990 年，Margaret Ellis 和 Bjarne Stroustrup 合著了 *The Annotated C++ Reference Manual*（也称 ARM）[Ellis90]。同年，ISO/ANSI C++标准委员会继续进展，并将 ARM 选作阶段的“起点”基础文档。ARM 是第一份包含模板描述的 C++参考，其中描述的模板并非今天所熟知的样子，那时的模板还相当单薄，整个规范说明和描述只占了 10 页纸。

  那时，人们关注的重心完全落在如何支持参数化类型以及参数化函数，ARM 中给出的例子是一个能够存放各种类型对象的 List 容器，以及一个 sort 算法，它可以对各种类型的序列进行排序。然而，即便是在那时，人们已经思考是否给模板一个分离式编译能

力。Cfront（Stroustrup 的 C++编译器）就支持当时尚处于初级阶段的模板的“分离式”编译，尽管它当时采用的做法并不具有可伸缩性（参见上一条中的标注）。

- 从 1990 年至 1996 年六年间，C++编译器开发商纷纷涌现，并各自为战采用不同的途径实现模板这一语言特性，同时标准委员会也对模板进行了极大的改进和扩充（使其更为复杂化了）。在透彻讲述 C++模板的著作 *C++ Templates* [Vandevoorde03]中，仅标准 C++模板的完整描述就占了 552 页中的大约 133 页篇幅，而整本书则完全用来详细描述模板这一语言特性以及如何有效地使用它。
- 在 20 世纪 90 年代的早期至中期，C++标准委员会将主要的精力放在让模板更健壮和实用，从而对它的基本使用有更好的支持。很少有人会怀疑他们创造出了一个极度灵活而有点古怪的奇迹般的东西，后来人们才知道模板原来本身就是一门图灵完备（Turing-complete）的元语言，用它可以写出任意复杂的程序，而这个程序是在编译期执行的。然而，如今在 C++中大行其道的模板元编程以及高级的库设计技术却是当初赋予 C++模板生命的人怎么也没有想到的，正是他们当时的决策造就了今天的一切，在 1990 年至 1996 年之间这些模板技术绝大部分还没出现呢！还记得吗，1994 年底 Stepanov 把他的 STL 库第一次提交给标准委员会，1995 年 STL 被采纳为标准，当时人们认为它是突破性的成就，然而在今天看来 STL“只不过”是一个容器和算法库而已。当然，STL 在 1995 年的背景下的确是个革命性的突破，而且即便是放在今天也仍然是令 C++区别于其他语言的强大证据之一，然而从如今的标准来看，STL 却只不过是 C++模板的牛刀小试而已。

70

这正是为什么我会说“在 1995 年至 1996 年间，模板本身仍然算是相当新颖的事物”的原因。如今 C++中的模板几乎可以算是最终形式了，然而即便是它的创造者当初也并没有完全意识到它们所能带来的影响。当时，整个 C++界的规模比今天小得多，鲜有编译器具有支持 ARM 中描述的模板之外的能力，大多数编译器对模板的支持都很单薄，甚至根本没有用。举个例子，那时只有一种商用编译器能够支持最初的 STL 实现。

因此，C++界尤其是 C++标准委员会对出现的简单 ARM 模板的实际经验仍然是比较缺乏的。1996 年整个 C++界已不能算处于萌芽期了，但还是比较年轻，处于不断成长和成型之中。

而正是在这种氛围当中，在如此有限的经验条件之下，标准委员会不得不决定是否将 export 特性留在标准草案当中。

1996

到了 1996 年，虽然可据参照的信息仍然少得可怜，然而大家都已经知道，export 使许多专家感到紧张。尤其是它令所有的编译器开发商都感到了紧张。即便是 export 的支持者也只不过把它当成一个不得已的折中，并同时为它所带来的复杂性深感不安。他们中有些人更喜欢无需任何特定关键字支持的一般性分离式编译。

在 1996 年，C++标准委员会曾一度出现大伙齐心协力想要将“分离式”模板编译的概念抛

弃的状况（最终事情以妥协收场，export 关键字正是因此从那时起被加入语言的，export 至少告诉了编译器哪些模板是应当分离式编译的）。特别地，曾有一种论调认为，分离式模板编译模式从来也没有真正实现过，标准委员会对于这个语言特性能否像当初预期的那样工作也毫无概念。几个 C++开发商都曾实现过各种形式的模板源代码组织模式，然而 export 与它们一概无关，export 是一个全新的、实验性质的特性，其背后没有任何的实现经验积累。事实上，当时曾有人就 export 发表了一些文章，详细讨论了标准草案中描述的 export 模式的主要潜在缺点，这些文章在现在看来可以说是具有洞察力和远见的。 71

特别地，当时所有的编译器实现方一致反对将分离式模板编译加入标准，他们认为这么做还过早，以至于没法判断这样做到底对不对。他们还对该特性当时的形式有着很深切但没有答案的关注和担忧（到底要不要 export 关键字），同时也觉得并没有足够的经验实现出一个成熟的替代方案（而且当时时间仓促——C++标准已经处于稳定状态，次年即 1997 年就将定案）。正是基于这些原因，所有的编译器开发商都一直不愿意将模板的分离式编译匆匆纳入第一代 C++标准[C++98]中。更确切地说，他们是想花费一些时间来尝试正确地设计它，等到下一代 C++标准再决定是否加入该特性。虽然他们从原则上也喜欢分离式模板编译这个想法，但他们更感觉到 export 尚未经过锤炼，况且仍然不知道如何正确地实现它。

然而可惜的是他们都未能成功，export 最终还是留在了标准当中[1]。不过对于这个结果我们是无法过分指责什么的。正如我前面所总结的，标准委员会的大部分成员（虽然人数并不多）相信，颁布一个标准，却缺乏像 C 对普通函数的分离式编译那样的模板“分离式”编译，是不完备且尴尬的。几个编译器都已经试验了“分离式”模板编译的一些不同形式，而且似乎从原则上来说这是个好主意。因而根据一贯原则（而且这是个良好的原则，不容轻视），export 就被留在了标准草案中。

为了强调，请注意世界范围内的编译器开发商只是反对 export，而并不反对模板的分离式编译原则。他们只是觉得需要更多的时间来确信标准这么做是正确的。尽管当年（1996 年）在会议上投票赞成保留 export 的一些世界级专家现在也认为当初的做法是个错误，但当时人们的动机和意图是良好的。而且就算是现在，由于人们在首次实现 export 的编译器（Comeau 4.3.01，发布于 2002 年）处得到一些经验，因而也仍然希望 export 能够带来一些好处（就算不能带来当初预期的那些大优点）。

目前我们获得的经验

全世界惟一实现了 export 特性的是 EDG（Edison Design Group），他们反映说，就他们的经验来看，export 是至今最难实现的 C++特性，与他们实现过的其他三大主要特性（例如名字空间或成员模板）耗费的工作量不相上下。仅是 export 本身就耗费了不止三个人年来编码和测试，而且这还没算上设计工作的开销，而同样这三个人实现整个 Java 语言也只不过耗费了两个人年。 72

1. 当时争论相当激烈，双方持续拉锯似的难分高下。在 1996 年 3 月的会议上，投票结果是 2 比 1 偏向于反对方。而在同年 7 月的会议上，export 关键字被引入了，结果投票最终又以 2 比 1 眷顾了 export。

那么，究竟为什么 export 这么难于实现，这么复杂呢？下面是两个主要原因：

（1）export 依赖于 Koenig 查找。大多数编译器在单个编译单元（其实就意味着一个源文件）中尚不能正确实现 Koenig 查找。export 要求跨编译单元实施 Koenig 查找。（参见[Sutter00，第27条]更多关于 Koenig 查找的讨论）

（2）从概念上来说，export 要求编译器同时处理多个符号表。实例化一个 export 的模板可能会触发其他编译单元中一系列的实例化，而当这个 export 模板的定义被解析的时候，这一系列的实例化必须都能够代表真实存在（或“某种意义上存在”）着的实体。在 C++中，处理单个符号表已经是够复杂的了，再加上 export，那么至少从概念上来说需要同时处理任意数目的符号表。

改……改……改……改变：export的阴影落在现有的语言特性之上

2. export 以何种方式改变了 C++语言其他特性的基本意义？简要地解释它们之间的相互影响。

export 对某些现有的语言特性具有一些惊人的影响。标准中对 export 的这些实际影响并没有提及或解决。尤其是，export 其实不止“导出”了它所标记的模板。

- 一旦有了 export，那么 export 模板所使用到的那些位于匿名名字空间中的函数和对象，现在就必须能够被跨编译单元访问和调用到了。类似地，文件作用域级的 static 函数和对象如果被 export 模板使用了的话，也就必须具有外部连接，或者至少得看上去具有外部连接。这跟匿名名字空间和名字空间范围内的静态对象的意图是背道而驰的，因为它们原本的意图是将这些名字严格限制在它们所处的编译单元中。（文件作用域级的 static 函数和对象是不推荐使用的，你应当使用匿名名字空间，然而它们仍然还是标准 C++的一部分。）
- 此外，有了 export，重载决议必须能够对来自任意多不同编译单元中的名字进行决议，有意思的是，这包括来自任意多个匿名名字空间中的重载名字。将“内部使用”的函数放入匿名名字空间中或使用不被推荐的 static，其用意本就是要将这些函数“私有化”，从而你可以赋予它们简单的名字，而不用担心名字冲突以及“跨源文件重载”。然而现在，由于这堵围墙已经为 export 开了一个口子，所以它们互相之间变得可见了，都参与了重载决议，这都得“归功”于 export。所以，如果某个内部名字被 export 模板使用了的话，你最好重新将它“修饰”一番[1]，即便这些名字是在匿名名字空间中或者是 static 也一样，只有这样做才能确保它们的意义不会被悄无声息地抹杀掉。

73

- 有了 export 还会带来新的二义性以及可能违反惟一定义规则(ODR)。例如，一个类可能在不同的编译单元中有多个友元实体，而位于这些单元中的该类的声明可能会同时参加一次实例化。如果出现这种情况的话，哪一组访问规则才是应当被采用的呢？这些问题可能看起来不太起眼，似乎有些错误也不会带来多大的危害，然而在某些常见的平台之上，ODR 的违反变得越来越值得重视（见[Sutter02c]中的一个例子）。

1. 例如，像 C 里面那样，加上类似“xxx_”的前缀。——译者注

export难于正确使用

3. export 对程序员有何影响？

跟普通模板相比，export 或许更难于正确使用。这里有三个例子可以说明这一点。

例 1：有了 export 的存在，程序员更是一不小心就会写出含义难以预测的程序了。被导出的模板跟包含模式的模板一样，也可以通过不同的路径被实例化，每条路径通常都意味着一个不同的上下文。有人可能会说："但是我们已经碰到过这种问题了呀，定义在头文件当中的函数（例如内联函数）不就是这样的吗？"但是请注意，这里我们面对的是模板，而当涉及到模板的时候这个问题就严重得多了，因为模板可能改变更多名字的意义，尤其是考虑到模板所使用到的名字比普通函数所用到的要广泛得多。模板使用了依赖名字，即那些依赖于模板实参（因而随之改变）的名字，因此对于同一组模板参数多次实例化某个模板的情况而言，模板的使用者必须小心确保其每次实例化点都具有同样的上下文（例如，能够接受某个模板实参类型的对象的重载函数集），以防止一不小心令模板的同一实例在不同文件中具有了不同的意义，造成典型的违反 ODR 的情况。那么，为什么我们会认为这个问题在 export 模式下比在包含模式下更恶劣呢？要知道，关于 export 的一个非常重要的地方就是除了模板通常具有的复杂性之外，export 使得来自多个编译单元中的名字都位于名字查找的"势力范围"之内了，而在 C++标准中的其他任何地方都未曾出现过这种情况。

例 2：export 使得编译器难以生成高质量的诊断信息来帮助程序员纠错。模板错误信息本身就因含有难以理解的冗长名字而声名狼藉，此外，程序员不太注意的是，对于编译器厂商来说，为模板提供良好的错误信息本就已经够困难了，因为某个模板的实例化可能会引发一系列瀑布式的实例化。而现在又有了 export，于是实例化的扩张链上又增加了一维——来自多个编译单元的实例化。结果我们所司空见惯的下述错误信息："第 X 行错误，由该函数的实例化导致，由该函数的实例化导致，由该函数的实例化导致……"现在就必须得在每级实例化前面加上它所属的编译单元的名字，而且这个回溯实例化过程的每一层实例化都可能是来自不同的编译单元。诚然，对于编译器来说，如何检测导出的模板是否违反了 ODR，本身也是一个富有挑战性的问题，然而要想"嗅"出程序代码的意图，以便能在提供错误信息的时候给出一点提示，则更是难上加难。许多人只盼望编译器能够为一般的模板提供可读一点的错误信息就心满意足了。

74

例 3：export 给构建环境也带来了新的约束。有了 export 的话，构建环境便不再单纯由.cpp 和.h 文件组成，今天的许多工具根本就不知道，在连接器（linker）可以回过头去改变.obj（或.o）文件的情况下[1]，如何处理哪怕是明显的循环依赖问题。正如前一条所指出的，如果你改变了一

1. export 的实现一般要求连接器（linker）能够回调编译器来进行实例化。这是因为在分离式编译的前提下，传统编译器是看不到其他编译单元的，而被导出的模板的定义体却位于其他编译单元之中，因此，编译器在编译那些使用了被导出模板的客户代码时，并不即刻进行实例化（因为编译器的视野并不能"穿透"编译单元，即编译器这时并不能看到被导出模板的定义，只能看到其头文件中的声明，因而无法进行实例化），而只是创建一个符号提示（hint），让连接器在连接期能够知道哪些符号是待决议的，从而回调编译器去完成相应的实例化。——译者注

个被导出的模板的实现文件，就需要将它重新编译一遍，而且不仅如此，你还要将对它进行实例化的客户代码也重编译一遍。也就是说，export 实际上并没有隔离依赖性，而只是将其隐藏了起来而已。

EDG 的 John Spicer 是模板技术的世界级专家，他曾经说过："export 从本质上来说是复杂且难以理解的，而且要想搞清它所带来的后果，需要相当大量的工作。对于 export 来说，我们很难给出浅显易懂的使用原则使用户免遇麻烦。"

export的潜在优点

4. export 究竟有何实际的潜在优点？

既然我们现在总算第一次拥有了可用的 export 实现，那么 C++界中当初支持该特性的家伙该可以捋起袖子去看看这玩意到底好不好用了。下面就是当初它的支持者期望它具有的两个实际和潜在的价值。

（1）*构建速度*。在现实编码当中，export 会对使用了模板的代码的构建速度产生怎样的影响（如果有影响的话）仍然是个未知数。如果该特性得到了广泛采用的话，在这个领域做些探索，我们会发现它所带来的优点出现得有多普遍，以及要想利用起这些优点我们还要做多少努力。特别地，人们期望那些使用了被导出模板的编译单元（跟使用一般包含式模板的编译单元相比）对被使用模板的定义改变较不敏感（即当被使用的导出模板的定义改变时，使用它们的编译单元重编译的代价较小）。

告诫：如果你想知道为什么 export 不能隔离依赖性，或为什么依赖性依然存在的话，请参考上一条。同样要注意，在 EDG 的模板实现当中，该潜在优势（即构建速度）不仅存在于 export 源代码组织方式上，而且还存在于一般的包含模式模板上，这意味着至少对于该实现（目前惟一的 export 实现）来说，export 跟包含模式相比并没有更多的优势。

75

（2）*宏渗漏问题*：这倒是 export 的一个实际优势。在传统的包含模式下，某些宏可能会渗漏进模板头文件中。由于包含模式模板的源代码对于每个使用它的编译单元来说都是完全可见的，所以其头文件外部的宏如果位于头文件#include 指令之前的话，就可能会对头文件中的模板定义产生影响。而有了 export，再加上宏并不能渗漏过编译单元的壁垒，这样模板代码的作者就能够对他们的模板定义（位于单独的源文件（编译单元）之中）有更好的掌控，并防止外部的宏对模板内部定义产生干扰。

防止宏渗漏是 export 的一个实际好处，然而并不是只有 export 才能提供这个好处。在 C++标准委员会早期在 C++0x 方面的工作中，EWG（Evolution Working Group，专门负责语言核心进化方面的工作）已经对宏的这个问题制定了更好、更通用的解决方案，适用于任何情况，例如 Stroustrup 的建议是加入可能是全新的#scope 和#endscope 预处理扩展指令。这类方案如果得到采纳的话，将会使 export 的这个优势完全消失，因为预处理域控制方案所带来的宏保护方面的能

力将完全超过 export 的，而且更为妥善，更为通用。

总之，我们仍需要在将来的几年里继续观察 export 跟通常的包含模式相比能够带来哪些额外的好处，不过我还是强烈建议那些对此进行测试的人，同时也去完整利用一下 EDG 实现的非导出模板优化能力，以作为对照，看看 export 究竟还剩多少额外的优点。

教训

那么，你究竟是否应当使用 export 呢？如果使用的话，又当如何安全地使用它呢？呃……就目前的情况来看，只有小部分 C++程序员会使用支持 export 的编译器来进行试验。对于绝大多数 C++程序员来说，关心是否使用 export 的问题是毫无意义的：他们手头没有相应的编译器，至少不会立即就有，因此他们无法使用它。

那如果你正在使用的恰恰是很有前途的、新式的、支持 export 的编译器，那又如何呢？啊哈，现在我们终于可以给出一个准则了：

准则：如果你想编写可移植的代码，那就别用export。

这似乎是不言而喻的：export 当然不能用在可移植的代码当中，因为如今的编译器对它的支持太贫乏了，因此，目前任何使用了 export 的代码从实际而言都是不可移植的，这种状况得持续一段时间。

76

然而，如果你并不需要可移植的代码，再加上你手头恰好有一个支持 export 的编译器，而且你很想去使用它的话，那该怎么办呢？如果真是那样的话，就得注意以下几个方面：记住 export 仍然是处于试验阶段的特性，而且它并没有带来人们所预期的好处，反而给 C++语言已有的其他特性带来了一些小问题。注意，编写导出模板的时候可能也会遇到一些棘手的问题，原因在这两条中和刚才的小结里面已经讲得很清楚了。

就目前而言，我能给出的最好的建议是：即便你现在只使用惟一一种编译器，而且它支持 export，通常你也应当尽量避免在产品代码中使用 export，因为这还是一个实验性的特性，让其他人冲锋陷阵去吧，因为我们将会在接下来的一两年中试验该特性，并弄清 export 真正能给我们带来什么。

准则：（就目前而言）避免使用export。

然而，如果你下定决心要身先士卒的话，下面是一些你可以做的事情，能让你的试验更安全，也更轻松。

准则： 如果你打算有选择地针对某些模板使用export的话：

不要以为export可以让你不用发布源代码（或其等价形式）。那是个误会，而且这一点永远也不会改变。

不要期望export会给代码的构建过程带来令人惊讶的速度提升。虽说最初的经验并不能说明什么，但你的构建时间反而可能会变得更长。

确信你手头的工具和环境可以应付新的构建要求和依赖性（例如，如果你的export实现是让连接器去改变它所接收到的.obj/.o文件的话，那么得确保你所有的工具都能够理解这一点）。

如果你的导出模板使用了任何来自匿名名字空间中的函数或对象，或者使用了任何文件范围内的static函数或对象的话：

- 必须认识到，这些函数/对象的行为就好像它们是被声明为extern似的，如果是个函数的话，它就有可能参与到来自任意数目的源文件中的其他匿名名字空间中的任意数目的函数重载决议当中去。

77

- 因此你应当总是将这些函数的名字加上额外的修饰（丑化，例如加上类似“xxx_”这样的前缀）以防它们无意中遭受语义改变。（这个结果令人遗憾，因为匿名名字空间和文件范围的static修饰的目的正是为了防止这种情况的发生，然而只要你使用了export，就很容易失去这种保护，因而你不得不重新将它们的名字修饰一番）。

认识到上面这些并非全部，你或许还会遇到一些其他的问题，而这些问题很可能超出了我们在通常的模板使用中积累的经验和认识。正如Spicer指出的：“很难单凭一些简单的准则就能够使我们免于麻烦。”要深刻认识到export某种程度上仍然还是一个试验性的特性，而且C++界目前尚没有机会来学习如何去使用export，因此目前我们尚未能够列出一些让你可安全使用export的良好原则。在不久的将来这种状况或许会有所改观。

现在断言说“避免 export”准则会永远正确还为时过早。时间和试验会说明一切。在接下来的几年中，随着编译器厂商逐渐（缓慢地）开始接受 export 并支持这一特性，我们将会逐渐了解到何时以及如何使用它，或者干脆不使用它。

78

异常安全问题及相关技术

在 C++等现代语言当中，异常处理是一项基本的错误报告机制。我们在 *Exceptional C++* [Sutter00]和 *More Exceptional C++* [Sutter02]中详细探讨了跟异常安全性定义相关的许多问题，以及如何编写异常安全的代码，需要注意的一些语言问题和语言特性间的相互作用问题。

本部分我们仍旧由这些材料出发，不过这次我们将注意力集中在一些特定的、与异常相关的语言特性上。我们先看一些老生常谈的问题：异常安全仅仅意味着在正确的位置写 try 和 catch 吗？如果不是，那又是什么呢？当你在为软件确定异常安全策略的时候，你应当考虑哪些东西？等等。

除此之外，我认为值得用一整条的篇幅来说明编写异常安全的代码对你来说只有好处没有坏处，因为这对你的编程风格是一种莫大的改善，从而会带来更健壮、更具可移植性的代码，更何况倘若异常真的发生了呢？然而，凡事都有个限度，古语云："过犹不及。"在我们了解了异常规格之后，这个限度就体现出来了：为什么语言中会有异常规格这一特性呢？为什么说它们的动机从原则上来说是良好的？为什么我们在写程序的时候应当尽量避免使用它们？

当然，除此之外还有一些其他的东西，现在就让我们上路，去聆听当今英才们的智慧吧。

第 11 条 try 和 catch

难度：3

异常安全仅仅意味着在正确的地点编写try和catch吗？如果不是的话那又是什么呢？当你在为软件制定一个异常安全策略的时候，应当考虑哪些东西？

JG问题

1. 什么是 try 块？

Guru问题

2. “编写异常安全的代码基本上即意味着在正确的位置写 try 和 catch”。请就此进行讨论。

3. 什么时候应当使用 try 和 catch？什么时候不应当使用它们？给出一个能够作为良好的编码标准的答案。

解决方案

进行catch

1. 什么是 try 块？

一个 try 块就是程序尝试去执行的一段代码（一组语句），try 块后面会跟有若干个异常处理块，如果 try 块中发生了异常，程序执行流就会进入相应的异常处理块中捕获相应类型的异常。例如：

```
// 示例11-1: 一个try块的例子
//
try {
  if(some_condition)
    throw string("this is a string");
  else if(some_other_condition)
    throw 42;
}
catch(const string&) {
  // 如果抛出的是一个string的话，执行这个处理块
}
catch(...) {
  // 如果抛出的是其他东西就进入这个处理块
}
```

80

在示例 11-1 中，被尝试执行的代码可能会抛出一个 string 类型的异常，也可能会抛出一个整型的异常，也有可能根本什么也不抛出。

除了进行catch之外还有其他东西

2. *“编写异常安全的代码基本上即意味着在正确的位置写 try 和 catch。”请就此进行讨论。*

坦率地说，这个说法代表了人们对异常安全的一种最为常见的误解。异常只是错误报告的另一种形式而已，我们当然知道编写错误安全（error-safe）的代码并不仅仅意味着在适当的地方检查**返回**码并处理错误条件。

事实表明编写异常安全的代码绝不仅仅意味着编写 try 和 catch，甚至这种做法越少越好。同样，始终牢记，异常安全性会影响一段代码的设计，不要以为你可以事后再去添加一些 catch 块。

在编写异常安全代码的时候主要得考虑三点因素：

（1）*应当在何时何地**抛出异常**？*这个问题是关于如何在正确的地点**抛出异常**的。特别地，我们得回答下列问题：

- 哪些代码应当**抛出异常**？也就是说，哪些错误值得我们使用抛出异常的方式来加以报告，而不用返回错误值或其他方式？
- 哪些代码不应当**抛出异常**？特别地，哪些代码应当提供“不失败”保证？（见第 12 条和 [Sutter99]）

（2）*应当在何时何地处理异常？*只有这个问题与“在正确的地方编写 try 和 catch”搭得上边，但甚至就连这个问题多数时候也是自动处理的。首先，考虑我们需要回答的问题：

- 哪些代码可以去**捕获异常**（catch）？换句话说，哪些代码具有足够的上下文和知识去处理给定异常所代表的错误（其处理异常的方式可能是将异常转换为另一种形式）？尤其要注意的是，负责捕获异常的代码同样需要足够的知识来进行任何必要的清理，例如对一些动态分配的资源的清理。
- 哪些代码应当去**捕获异常**？换句话说，在可以用于捕获给定异常的代码当中，哪个才是最合适的？

81

在解答了以上这些问题之后，我们应当注意到，使用“资源获取即初始化（RAII）”惯用法，可以消除许多不必要的 try 块，因为该惯用法能够使异常环境下的清理善后工作自动化。如果你将动态分配的资源包装在一个对资源具有所有权的管理类对象当中的话，通常不管异常发生与否，该类的析构函数都会忠实地在正确的时机进行自动的资源清理，从而你根本就无需编写 try 或 catch 块。这种能力显然让人感到很舒服，更何况这种做法更容易编码，也更便于日后的阅读。

准则：尽量通过析构函数来进行异常环境下的自动清理工作，而不是通过try/catch。

（3）*在其他所有地方，假设被调用的任何函数都会抛出异常，那么我们的代码是否仍然安全呢？*这一考虑促使我们去使用良好的资源管理技术来避免泄漏、维持类或程序的不变式（invariant）以及一些其他影响程序正确性的问题。从另一个角度来说，这是为了防止仅仅由于某个异常碰巧在从它的抛出端传播到处理端的途中经过了某些根本无需特意关心该异常的代码而

导致整个程序的崩溃。就我所遇到的大多数程序员来看，似乎这是到目前为止关于异常安全的最耗时间、最难弄懂的一个方面。

注意，在以上三个问题之中，只有一个问题是跟编写 try/catch 有关的。而且即便是那个问题也可以通过聪明地使用析构函数进行自动清理从而回避掉。

3. 什么时候应当使用 try 和 catch？什么时候不应当使用它们？给出一个能够作为良好的编码标准的答案。

下面就是一则建议。我以简洁的形式描述一下：

（1）为你的程序或子系统确立一个整体上的错误报告/处理策略并始终遵循它。特别地，该策略应当照顾到如下几个基本的方面（通常就我们遇到的实际情况而言，远远不止要考虑这些方面）：

- 错误报告。对哪种错误需要被报告以及如何报告它们给出一个明确的定义，尽量使用异常而不是其他错误报告机制。通常，对你面临的每种情况都默认选用最具可读性和可维护性的方案，例如，对于那些不能返回值的函数（如构造函数和操作符）来说，异常是最有效的方案。同样，对于错误抛出端与处理端相隔遥远的情况来说，异常也是最理想的方案。
- 错误传播。明确定义异常不应当穿越哪些边界，通常这就是指模块或 API 边界。
- 错误处理。规定在任何可能的情况下使用对资源具有所有权的对象及其析构函数来执行清理任务。

82

（2）只在那些侦测到错误但自己却无法处理的地方抛出异常。（很明显，能够就地将错误处理掉的代码当然不需要报告该错误了！）

作为每个函数或模块的文档的一部分，在每个操作的文档里面明确指出该操作可能会抛出哪些异常，以及为什么抛出这些异常。你并不需要真正去为每个函数加上异常规格声明（事实上不应该这么做，见第 13 条），但确实有必要在文档中明确且严格地指出函数的调用方可以对它的异常语义作什么样的假设，因为错误语义也是一个函数或模块接口的语义的一部分。

（3）在设置 try 和 catch 的地方必须有足够的知识和上下文去处理/转换特定异常，或强制实施错误策略中指出的异常边界。我发现编写 try 和 catch 有三个主要理由：

- 为了处理错误。这是最简单的情况：错误发生了，我们知道应该对此做些什么，于是就做了。程序继续运行（原先的异常已经被安全地处理掉了）。再次申明，尽可能地在析构函数中完成你的动作，实在不行才去使用 try/catch。
- 为了转换异常。这意味着捕获一个汇报了较低层错误的异常，然后抛出另一个在执行转换的代码自身的（较高层面的）语义上下文中重新表达的异常。还有一种做法是将原先的异常转换为另一种表示形式，如错误代码。

  例如，考虑一个通信会话（session）实用类，它能应付多种主机（host）类型和传输协议，而打开与另一个主机会话的尝试可能会因任意的低层原因而失败，失败原因能够被会话类探测出来（例如无法找到网络或远程主机拒绝认证/许可等）。Open()函数自身就

足以应付这些情况，因而没必要将它们汇报给调用方，毕竟调用方对会话中涉及的具体哪个分组一无所知，也不知道该分组出现某种情况时该怎么办。会话类直接就将它的内部发生的低层错误处理掉了，从而令它自身仍然处于一致的状态下，并同时汇报一个属于会话类自身的、较高层别的错误或异常来告知调用方“本次会话无法打开”这一信息。

```
void Session::Open(/*...*/) {
 try {
  // 整个的操作
 }
 catch(const ip_error& err) {
  // - 对IP错误进行一些处理
  // - 清理
  throw Session::OpenFailed(); // 抛出另一较高层的异常——译者注
 }
 catch(const KerberosAuthentFail& err) {
  // - 对认证错误进行一些处理
  // - 清理
  throw Session::OpenFailed(); // 抛出另一较高层的异常——译者注
 }
 // ……其他内容……
}
```

83

- 为了在子系统边界或其他运行时防火墙边界上进行 catch(...)。这通常也会涉及到异常/错误的转换，通常是转换为错误代码或其他非异常形式的表示方式。例如，当栈展开（stack unwind）到达了某个 C API 的时候，就只剩下了两个选择：一是从该 API 函数中立即返回一个错误代码，二是设置一个错误状态以便事后调用方可以通过 GetLastError() API 函数去查阅。

准则：为应用程序或子系统确立一个整体的错误报告/处理策略，并始终遵循它。并为错误报告、错误传播以及错误处理制定策略。
在那些侦测到错误而自身又无法对其进行处理的地方抛出异常。
在那些具有足够知识和上下文信息去处理/转换错误或强制实施边界条件的地方编写 try/catch（例如，在子系统边界上或其他运行时防火墙边界上进行 catch(...)）。

小结

一位智者曾经说道：

要么领路，要么跟从，否则闪到一边去！

而在异常安全分析中，我们也许同样可以这么说：

try、catch，*否则闪到一边去！*

在实践当中，最后一个“闪到一边去”占有了异常安全分析和测试的大部分份额。这正是我为什么会说异常安全编码并不仅仅意味着在正确的位置编写 try 和 catch 的原因。进一步说，异常安全编码通常即意味着在正确的地点避开“路过”的异常。

84

第 12 条 异常安全性：值得吗

难度：7

费力去编写异常安全的代码值得吗？现在这个问题应当不再有什么异议或争论了……然而，有时候仍然需要加以考虑。

Guru问题

1. 回顾：简要地定义 Abrahams 异常安全保证（基本保证、强保证以及无失败保证）。

2. 什么时候值得去编写满足下列异常安全保证的代码？

（a）基本保证

（b）强保证

（c）无失败保证

解决方案

Abrahams保证

1. 回顾：简要地定义 Abrahams 异常安全保证（基本保证、强保证以及无失败保证）。

基本保证即是指失败的操作可能会改变程序的状态，然而不会发生任何泄漏，受失败的操作所影响的对象/模块仍然是可析构且可用的，并处于一致的状态下（不过该状态并不一定是可预测的）。

强保证意味着事务提交/回滚语义：从被操作对象的角度来说，操作即便失败也不会导致程序状态的改变。这意味着失败的操作对被操作对象不会产生任何的副作用，包括那些相关的辅助对象（例如指向被操作容器内部的迭代器）的内容和有效性也不会受到影响。

最后，无失败保证是说根本不允许失败的发生。从异常的角度来说就是指某个操作绝不会抛出异常。（Abrahams 以及其他一些人，包括我在 *Exceptional C++*书中，都将该保证称为不抛出保证。）是我将它改称为无失败保证的，因为该保证对于所有的错误处理策略都同样适用，不管该错误处理策略是基于异常还是基于一些其他的机制（如错误代码）。

85

什么时候值得让你的代码具有更强的异常安全保证？

2. 什么时候值得去编写满足下列异常安全保证的代码？

（a）基本保证

（b）强保证

（c）无失败保证

始终要让代码至少满足以上保证之一。有如下几点很好的理由：

（1）异常总是会发生的（借用一句老话——天有不测风云）。你就是摆脱不了它们。标准库函数可能会抛出异常；语言实现也可能会抛出异常。我们得对这些意外情况有所防备。所幸的是这并不算什么大问题，因为我们现在已经知道怎么正确地应对了。然而，这倒是要求我们首先得接受一种新的习惯，并持之以恒（对于学习针对错误代码的编程也同样如此）。

棘手的问题还是关于错误处理的一个一般性问题。即从细节上来说，到底是使用返回错误码还是抛出异常的形式来报告错误，几乎只是个语法细节上的差异而已，主要区别在语义上，即错误报告是如何完成的，因而每种做法都需要其自身的风格。

（2）编写异常安全的代码是有好处的。异常安全的代码往往就意味着良好的代码。被普遍用来帮助人们编写异常安全代码的技术，通常也正是我们即便在基本无异常的环境下也应当使用的技术。也就是说，即便你无需考虑异常安全性，支撑异常安全编码的技术本身对你也有利无弊。

为了实际地来考察这个问题，考虑下面一些使异常安全变得更容易的主要技术，我跟其他一些人都曾写过这方面的文章：

- 使用“资源获取即初始化”（RAII）惯用法来管理资源的所有权。使用对资源具有所有权的对象，如 Lock 类和 shared_ptr 类（见[Boost,Sutter02a]），通常是个好主意。我们会毫不惊讶地看到，它们除了具有很多优点之外，还具有异常安全性。你有多少次见过这样的情形：某个函数（当然我们是在说别人写的函数，你才不会写出这样的函数呢）由于没有使用 RAII 手法来自动管理资源，同时其函数体中的某个代码分支却又导致了过早返回（return）从而没能进行资源清理，结果导致资源泄漏[1]。
- 使用“先在一旁将所有事情做完，然后再通过不抛出异常的操作来提交整个任务”的手法，避免在不确定所有操作能否都成功的情况下贸然去改变对象的内部状态。这种事务性编程清晰、干净，也比较安全，即便是使用错误码来报错也一样。你又有多少次见过这样的情形：某个函数（当然我们还是在说别人写的函数，你才不会这么写函数呢）由于并没有采用上面所说的手法，然而其函数体中的某个代码分支却又导致了过早返回，

86

1. 参考这样一段代码：

```
void may_leak_mem()
{
  int* p = new int;
  if(...)
  {...}
  else
  {...; return;}        // 如果执行流进入这个分支，便会导致所谓的
                        // 过早返回，导致p得不到释放。
                        // 使用RAII可以很轻易地解决这个问题
                        //（例如，将p交给一个auto_ptr）。
  delete p;
}
```

此时由于在失败的操作之前对象的内部状态已经部分地改变了，因而这里的过早返回就导致对象被留在了不一致的状态下。

- 尽量使用“单一职守的类（或函数）”。像 Stack::Pop()和 EvaluateSalaryAndReturnName()（分别在 *Exceptional C++* [Sutter00]的第 10 条和第 18 条介绍）这样的做多件事情的函数是很难实现强异常安全保证的。只需简单地遵从单一职守原则，许多异常安全问题都会变得更简单，甚至无需多加思考就迎刃而解了。而且单一职守这一原则很早就出现了，只是后来我们才发现它恰巧能够用在解决异常安全问题上，其本身就是一个很好的原则。

遵循上面这些原则对你只有好处没有坏处。

话虽如此，我们到底什么时候该使用哪种异常安全保证呢？简单地说，下面就是 C++标准库所遵从的原则，你可以有效运用到自己的编码中去：

准则：函数应当总是支持它所能支持的最强的异常安全保证，但前提是不能给那些并不需要该保证的调用者带来额外开销。

这就是说，如果你的函数可以支持无失败保证，同时又不会给不需要该保证的调用者带来额外的性能开销的话，你就应当让它具有无失败保证。但也要注意，少数关键的函数一定得是无失败操作：

准则：永远不要允许析构函数、释放操作（deallocation）以及swap()函数抛出任何异常，因为否则的话，就没法安全且可靠地进行资源清理了。

否则，如果你的函数可以支持强异常安全保证，同时又不会给不需要该保证的调用者带来额外开销的话，你就应当让它具有该保证。注意，vector::insert()是一个有趣的例子：它通常并不支持强异常安全保证，因为这样做意味着每次我们插入一个元素的时候，它都得对整个 vector 的内容进行一份完整的副本才行，况且也并非所有的程序都如此在乎一个需要付出这么大代价才能换来的强异常安全保证。（那些在乎这一点的程序员可以手动将 vector::insert()包装成一个具有强异常安全保证的操作，很简单：首先对 vector 作一份副本，然后对这份副本进行 insert()，一旦这个插入操作成功了，最后就将原来的 vector 跟这份副本 swap()一下即可。）

87 否则你的函数就应当支持基本的异常安全保证。

要想知道这些概念更为详细的信息，诸如“一个不抛异常的 swap()都意味着些什么”，还有“为什么析构函数永远不该抛出异常”等等问题，可参考 *Exceptional C++* [Sutter00]以及 *More*
88 *Exceptional C++* [Sutter02]。

第 13 条　对异常规格的实际考虑

难度：6

C++界已经在异常规格方面获得了一定的经验，所以现在正是时候反省一下究竟何时及怎样使用它们。该条考察了异常规格的有用之处及乏善可陈之处，以及现实中的不同编译器的实现对异常规格的可用性有哪些具体的影响。

JG问题

1. 违反异常规格会导致什么样的后果？为什么？讨论该 C++特性的基本理念。

2. 描述如下的函数可能抛出哪些异常。

```
int Func();
int Gunc() throw();
int Hunc() throw(A,B);
```

Guru问题

3. 异常规格是函数类型的一部分吗？请解释。

4. 什么是异常规格，它们是用来做什么的？请准确描述。

5. 什么时候值得为函数加上异常规格？为什么你选择这样做，或者为什么不呢？

解决方案

考虑到目前的工作正向着下一代 C++标准（即 C++0x）推进，所以现在正是时候来考察一下，我们在当前标准[C++03]下获得的经验告诉我们能做些什么，以及从这些经验当中我们学到了哪些东西。标准 C++当中的特性大多是良好的，而且它们在有关 C++的文献当中占去了最多的篇幅，因为人们对那些较为不起眼的特性觉得没什么好说的。更确切一点说，那些不起眼或不那么有用的特性更多时候只是为人们所忽视，并逐渐被尘封在语言的角落，直到许多人甚至于忘记了它们的存在（虽然这并非总是件坏事）。这正是为什么你看到的关于它们的文章总是相对较少，譬如像 valarray、bitset、locale，以及实际上是合法的表达式“5[a]”这些冷僻的特性（虽然本书的后面有一条是关于 5[a]的）。而对于异常规格来说同样如此。 89

现在，让我们来更细致地考察一下我们在标准 C++异常规格方面获得的经验。

违章

1. 违反异常规格会导致什么样的后果？为什么？讨论该 C++特性的基本理念。

异常规格的理念在于使用运行期检查来确保只有某些特定的异常能够从某个函数当中逃逸

出来（其他情况下则不会有任何异常从中逃出来）。例如，如下的函数的异常规格确保了 f 只会允许 A 或 B 类型的异常从它的函数体内逃出来：

```
int f() throw (A,B);
```

如果函数体内抛出了某个不在该函数的异常规格列表里面的异常，unexpected()函数就会被调用。例如：

```
// 示例13-1
//
int f() throw(A, B) {          // A和B都跟C无关
  throw C();                   // 会导致unexpected被调用
}
```

可以通过调用标准的 set_unexpected 函数，来注册你自己的用于处理非预期异常（unexpected-exception）的函数。你自己的处理函数必须为无参且返回 void 的函数。例如：

```
void MyUnexpectedHandler() { /*...*/ }
std::set_unexpected( &MyUnexpectedHandler );
```

剩下来的问题就是，这个用于处理非预期异常的函数究竟能够做些什么？首先，无论如何有一件事情它不能做，即它不能通过通常的函数返回方式来返回。而它可以做的事情有如下两件：

- 它可以决定将当前异常转换为某个被异常规格列表所允许的异常，具体做法是抛出它自己的、满足那个引发它被调用的异常规格列表要求的异常。于是栈开解（stack unwind）就会从原来离开（去调用该函数）的地方继续下去。
- 它可以调用 terminate，该函数会终止程序。（terminate 函数本身也可以被替换掉，但替换它的任何函数同样还是必须终止程序。）

90

到目前为止……

异常规格背后的理念很容易理解：在 C++程序中，除非一个函数带有异常规格声明，否则我们就认为它可能抛出任何异常。考虑这样一个名为 Func 的函数（由于名字 f 被用得太滥，所以这里我们不用 f）。

2. 描述如下的函数可能抛出哪些异常。

```
// 示例13-2(a)
//
int Func();  // 可以抛出任何异常
```

在 C++中，默认情况下 Func 实际上就像它的注释当中描述的那样，可以抛出任何异常。但实际上，通常我们知道一个函数可能会抛出哪些异常，所以一个相当合理的做法就是向编译器和程序员提供这方面的信息，确切地告诉他们哪些异常会逸出某个函数。例如：

```
// 示例13-2(b)
//
```

```
int Gunc() throw();                    // 什么也不抛出
int Hunc() throw(A,B);                 // 只能抛出A或B
```

在这些情况下，函数的异常规格就是在向外界宣告 Gunc 和 Hunc 可能会抛出哪些异常。而这两个函数后面的注释则用自然语言复述了异常规格的含义。我们待会儿再回到“用自然语言”这个问题上来，因为正如它们看上去那样，这两个注解给人一种正确的假象（而其实是错误的）。

人们自然可能会认为明确地表示出某个函数可能会抛出哪些异常是件好事，或者认为信息总是越多越好。这么想的人可能就错了，因为危险隐藏在细节当中：尽管人们的出发点是良好的，但 C++中异常规格的规则决定了它并非总是有用的，甚至往往是完全有害的。

问题一：一个“浅陋的类型系统”

3. 异常规格是函数类型的一部分吗？请解释。

John Spicer 是著名的 Edison Design Group 中的一员，曾撰写了 C++标准中关于模板的大部分内容，他把 C++的异常规格称为“浅陋的类型系统”。C++最强的特性之一就是其强类型系统，而且 C++在这方面确实表现得不赖。那为什么我们不把异常规格称为 C++强类型系统的一部分，却将它称为浅陋的类型系统呢？

91

原因很简单，分为两方面：

- 异常规格并非函数的类型的一部分……
- ……有些时候它们又的确是。

首先考虑下面这个例子，它展示的是何时异常规格并不影响函数的类型。思考下面的代码：

```
// 示例13-3(a): 你不能在typedef当中写异常规格。
//
void f() throw(A,B);

typedef void (*PF)() throw(A,B);          // 语法错误

PF pf = f;                                // 不能到达这里，因为上面有错误！
```

代码中的那个 typedef 中的异常规格是非法的。C++不允许这样写，因此，C++并不允许异常规格成为函数的类型的一部分……至少在 typedef 中不能。但在其他情况下，异常规格却又确实是函数的类型的一部分，譬如说，如果你把上面代码中的 typedef 关键字去掉，而其他部分保留原样的话：

```
// 示例13-3(b): 但如果把typedef去掉就可以了！
//
void f() throw(A,B);

void (*pf)() throw(A,B);                  // 没问题

pf = f;                                   // 没问题
```

对于函数指针赋值而言，只要赋值目标（即函数指针）的异常规格不比赋值源（即函数）的异常规格更严格，赋值就可以成立。

```
// 示例13-3(c): 同样可以，低碳、无脂[1]。
//
void f() throw(A,B);

void (*pf)() throw(A,B,C);              // 没问题

pf = f;                                 // 没问题，pf的异常规格比f宽松
```

此外，如果你想重写虚函数的话，异常规格也会影响虚函数的函数类型：

```
// 示例13-3(d): 异常规格会影响虚函数。
//
class C {
  virtual void f() throw(A,B);          // 跟上例中f的异常规格相同
};
class D : C {
  void f();                             // 错误，现在异常规格就开始发挥作用了[2]
};
```

92

所以对于当前 C++中的异常规格而言，其首要问题在于它们确实是个“浅陋的类型系统”，它们的游戏规则跟 C++类型系统的其他部分不同！

问题二：（错误的）理解

第二个问题：弄清楚你究竟能得到什么。正如许多著名的 C++专家（包括 Boost 库的异常规格理念[BoostES]的作者）所指出的那样，程序员倾向于使用异常规格，仿佛认定了它们会具有他们想要的行为似的，而事实却不然。

因此，我们给出另一个问题：

4. 什么是异常规格，它们是用来做什么的？请准确描述。

以下是许多人所认为的异常规格能做的事情：

- 确保函数只会抛出它的异常规格列表里面所列出的异常（也可能不抛出任何异常，即异常列表为空的情况）。

1. 原文是 kosher，有“合适”、“可以”的意思，其含义源于“符合犹太教所规定允许的饮食习惯”，后文“低碳、无脂”故作此意。——译者注
2. 派生类的虚函数必须拥有比其基类的对应虚函数更为严格的异常规格，这个限制很明显，一个派生类对象可以被当成基类对象来使用，所以派生类对象的虚函数也理所当然应该满足其对应的基类虚函数所具有的限制（这里的限制是异常规格）。站在使用指向基类的指针来操纵派生类对象的用户的角度来说，他们会假定所调用的 f 不管动态分派到哪个派生类对象上去，都满足静态决议出的 f（即基类的 f）所限定的异常规格，即至多只能抛出 A 和 B，这样他们的代码中就只需捕获 A 和 B 异常。而本例中的 D::f 却可能抛出任何东西，如果语言允许这一点的话，D::f 抛出的其他异常对于把它当成 B::f 的调用方代码来说就是漏网之鱼了！后果可想而知。——译者注

- 让编译器知道只有那些列出的异常（也可能没有）才可能被抛出，从而使编译器能够基于该信息来进行优化。

再一次提醒，上面的这些期望只是看起来正确而已。再次考虑示例 13-2(b)中的代码：

```
// 示例13-2(b) 重提一下前文的例子，因为这里有两个潜在的“善意的谎言”：
//
int Gunc() throw();                    // 什么也不抛出    ← ?

int Hunc() throw(A,B);                 // 只会抛出A或B    ← ?
```

以上代码中的注释正确吗？不见得。实际上 Gunc 函数体内可能抛出异常，而 Hunc 函数体内也可能抛出除 A 跟 B 之外的其他异常！编译器只是确保会将这些异常“镇压住”而已，如果它们胆敢“作乱”的话……哦，顺便提一下，编译器的这种做法大多数时候也会令你的程序“人事不省”。

由于 Gunc 或 Hunc 实际上可能会抛出它们承诺不会抛出的异常，所以不光是编译器不能假定这种事不会发生，而且编译器同样还要扮演警察的角色，四处巡视以便确保这类糟糕的事情不会在人们毫无觉察的情况下发生。如果它确实发生了，编译器必须调用 unexpected 函数。大部分时候这会导致你的程序终止。为什么？因为碰到了 unexpected 你只有两条路可走，不管哪条路都不是正常的返回。你可以根据具体情况进行抉择：

93

- 抛出一个异常规格列表允许的异常。如果采取了这种做法，异常就会按照通常的方式继续传播。但是，不知道你还记不记得，unexpected 处理函数是全局的。也就是说，对于整个程序来说只有一个 unexpected。全局处理函数几乎不可能智能到能够针对任意给出的特定情况作出相应的正确响应，所以结果它只能够调用 terminate 了事，而且是直接调用 terminate，不经过 catch，不拖泥带水[1]。
- 抛出（或重新抛出）一个异常规格列表（仍然）不允许的异常。如果原来的函数的异常规格列表允许 bad_exception 类型的话，没问题，现在就轮到 bad_exception 开始传播了。但如果不允许的话，结果就是 terminate，直接 terminate……

由于违反异常规格大部分时候最终会导致你的程序终止，所以我觉得将它的行为称作“把你的程序打得不省人事”是合适的。

前面我们曾经提到许多人所认为的异常规格能做的事情。下面我对他们的想法作了圈点，更

1. 这里作者说的是“go to terminate, go directly to terminate, do not pass catch, do not collect $200”，这句话源自一个名为垄断（monopoly）的非常经典的游戏。游戏场地是一块（或两块）方形木板，四周有一格一格的道路，玩家执骰子，然后根据骰子点数挪动自己的令牌，道路的每一格有指示以及财产等，玩家的目标是尽量利用现有设施（如旅店，停车坪等）向其他人收费以挤垮其他玩家，让自己成为最后一个不破产的人。而有一格上面的话就成了经典“go to jail, go directly to jail, do not pass GO, do not collect $200.”jail（监狱）也是棋盘上的一格，与该格呈对角势，该指示的意思是，直接进入 jail（监狱），不要一格一格挪动，不要经过“GO”格，不要收集 200 美元（玩家如果一格一格走可以收集到一定的钱币），与此句类似的还有其他出自该游戏的经典句子如“play my get-out-of-jail-free card”等。这些大概是美国儿童从小时候就耳熟能详的句子。——译者注

精确地描绘了异常规格到底真正能够做些什么[sic][1]：

- **强制**（Enforce）函数**在运行期**只能抛出其异常规格列表里列出的异常（也可能不抛出任何异常）。
- 编译器**必须检查是否的确只有列出的异常被抛出了**，这一点使编译器能够（**或阻止了编译器**）去进行优化。

为了弄清编译器究竟必须做什么，考虑如下的例子，它为我们的示例函数 Hunc 提供了一个函数体：

```
// 示例13-4(a)
//
int Hunc() throw(A,B) {
  return Junc();
}
```

从功能上说，编译器必须生成类似如下的代码，它的运行期开销跟你自己手写这些 try-catch 相比通常是一样的（虽然省去了许多键盘敲击，因为是编译器自动帮你生成的）：

94

```
// 示例13-4(b): 示例13-4(a)经编译器改过的版本
//
int Hunc()
try {
  return Junc();
}
catch(A) {
  throw;
}
catch(B) {
  throw;
}
catch(...) {
  std::unexpected();  // 不返回!但可能会抛出A或B，如果你幸运的话。
}
```

这里我们可以把在异常规格背后发生的事实看得更清楚，并不是假设只有某些特定的（异常规格列表中指出的）异常会抛出从而让编译器据此进行优化，而是恰恰相反：编译器必须得在运行期进行更多的工作以确保只有这些特定的（异常规格列表中指出的）异常会从函数中逃逸出来。

深入异常规格

大多数人在发现异常规格会导致性能损失的时候都会感到惊讶。导致这个结果的原因之一我们已经详细描述过了：异常规格会导致隐式生成 try/catch 块，从而带来额外开销，尽管在高效

1. 对，这是个 sic（原文如此）玩笑。

的编译器上这个开销可能比较不起眼。

异常规格通常至少有两种方式会导致运行时的开销：

- 有些编译器会自动拒绝将那些具有异常规格的函数进行内联，这种做法跟它们的其他启发式方法类似，譬如拒绝内联那些具有多于某个特定数目的嵌套语句，或者包含任何循环结构的函数。
- 有些编译器根本不能很好地基于异常相关的信息进行优化，对于这些编译器而言，即使函数体内的代码清清楚楚地表明不可能抛出任何异常，它们也会在里面生成 try/catch 块（这是事实，我并不是开玩笑）。

除了运行期开销之外，异常规格还会导致开发时间上的开销，因为它们增大了耦合性。例如，只需简单地从基类的虚函数的异常规格列表里面删除一个类型，就会发现许多其派生类都不能通过编译了，如果你想这么干的话，这是最简单而直接的方式。在星期五下午试一下这种做法，我敢打赌星期一早晨你会在收件箱内发现无数封愤怒的邮件在那儿等着呢。 95

因此我们的下一个问题自然就是：

5. 什么时候值得为函数加上异常规格？为什么你选择这样做，或者为什么不呢？

到写作本章为止，C++界得到的几乎最好的建议是：

准则： **原则1：永远不要为函数加上异常规格。**
原则2：除非你想声明的是空异常规格列表（即throw()），但如果是我的话，我甚至连这种做法也会避免。

Boost 库的经验是，惟一“可能在某些编译器上带来某些好处”的异常规格声明，就是为一个非内联函数加上“不抛出异常”的异常规格声明（即 throw()）。这个陈述本身根本不能激起我们的兴趣，但如果你必须编写可被用在不止一个编译器平台上的可移植的代码的话，这倒是个有用的思路。

然而，事实上，它甚至比实际情况中的还糟糕一些，因为似乎一些流行的编译器实现处理异常规格的方式各有不同，至少有一个流行的 C++编译器，即微软的 Visual C++（截止到本书的写作为止，最新的版本是 Visual C++ 7.1(2003)）仅仅只会去解析异常规格，而实际上却不去实施它，这就令异常规格变成了有名无实的注释。但是等一下，还有在同一时间，编译器却仍然可能会假定函数的异常规格在函数内部得以实施，从而基于这个假定在这些函数之外进行一些合法的优化（虽然这个假定可能并不成立）。Microsoft 7.x 编译器就会这么做。编译器之所以这么做的理念是：如果该函数确实曾试图抛出它不该抛出的异常的话，其内部的处理函数就会终止程序，从而控制流就不会返回到调用方，因此，既然我们是位于调用方的代码中，即假定控制流返回到了调用方，那么编译器在为调用方生成代码时就可以假定被调用的函数什么也没抛出来，从而做一些优化，譬如消除该函数外部的 try/catch 块。

这些编译器根本不去实施异常规格，但却仍然**依赖于**“它被实施了”这个假定，所以说，在这些编译器上，throw()的含义改变了，原来C++标准中的说法是“根据这个来检查我，如果我一不小心抛出了异常，那就终止我”，而在这些编译器上却变成了“相信我说的，假定我永远不会抛出，并据此进行你的优化吧。”所以你得小心了：如果选择使用了哪怕一个空的异常规格，也得阅读一下你的编译器的文档，看看它对此实际上会做些什么。你可能会大吃一惊。小心驾驶哦！

96

小结

一句话，别去惹异常规格。即便专家也是如此。

稍微详细一点说，主要问题在于：

- 异常规格可能会导致令人惊讶的性能损失，例如，倘若编译器对那些带有异常规格的函数不予内联的话。
- 对于异常规格意图捕获的错误来说，运行时的 unexpected 错误不总是你所愿意看到的。
- 通常不能为函数模板编写有用的异常规格，因为你无法预知它们所操作的类型会抛出什么。

这些材料是我不久前在一个会议上的演讲内容的一部分，我曾询问当时在座的大约100位听众当中有哪些人曾使用过异常规格。大概有一半人举手。然后后面有一个家伙说我应该询问到底有多少人后来又把异常规格给拿掉的（他说得很对），所以我就问了，同样差不多的人举了手。这就表示，Boost 库中的那些世界级的库设计好手们也曾有过同样的经历，这正是为什么他们在异常规格方面给出的编码策略几乎归结为“别那么做”[BoostES]的原因。

是的，许多人出于善意的目的想让异常规格进入语言，而这也正是它们现在位于语言当中的原因。这令我想起大约15年前的一个冬天在 email 上传播的一首诗。当时诗的名字为“临近圣诞节前夜”，如今我将它更名为“临近实现的前夜”，或者“临近危机的前夜”。说的是一个编程大师为了满足用户设的最后期限而苦苦奋斗至节假日的深夜，他完成了近乎不可能的任务，并做出了一个完美地实现了需求的可工作的系统……然而他最终得到的只不过是项目报告的最后四行，寥寥几笔措辞委婉地指出他工作的草率：

系统已经完成，测试也已经结束，
甚至连用户最近一次要求的改动都已经实现。
但用户却咆哮着大声奚落：
“这只不过是我们所要求的，而不是我们想要的！”

我们完成对目前在异常规格上的经验的思考之后，得到了同样的感觉。这个特性当时看起来似乎是个好主意……而且它确实是有些人所要求的。

97

诚心对待你的愿望，它就有可能实现。

类的设计、继承和多态

除了泛型编程之外，C++同样对面向对象的设计和编程具有良好的支持。本节我们就将焦点集中到这个较为传统的领域，并对 C++支持面向对象的方式加以特别的关注。

我们先看一个具有微妙缺陷的编码实例，以它为出发点来回顾基本的对象构造和析构顺序。接着我们继续深入，讨论如何编写健壮的代码，这就涉及了代码安全的问题：首先，类的哪些部分是可以被多方访问到的？特别是，哪些途径会“泄漏”一个类的私有成分（故意地或出于其他什么原因）？什么是封装？它跟我们对于成员可访问性所能做且应做的决策有什么关系？最后，怎样才能让我们的类避开版本问题，并确保基类合同的易维护性及不易被无意间破坏掉？比如对派生类而言，容易导致合同破坏甚至安全漏洞。

随着在对象的世界里不断深入，我们当然还会发现更多……

第 14 条　顺序，顺序！

难度：2

C++能干什么，不能干什么？程序员在学习C++的时候常常会对这个问题产生有意思的误解。下面这个例子是由Jan Christiaan van Winkel给出来的，展示了一名学生犯了一个基本错误，然而许多编译器却让程序安然通过编译，就连一个警告也不给。

JG问题

1. 以下代码实际上是一位学 C++的学生写的，该学生所使用的编译器对这段代码并不报警。实际上，就连好几个流行的编译器也是如此，那么下面的代码中有什么错误？为什么？

```
#include <string>
using namespace std;

class A {
public:
  A(const string& s) { /* ... */ }
  string f() { return "hello, world"; }
};

class B : public A {
public:
  B() : A(s = f()) {}
private:
  string s;
};

int main() {
B b;
}
```

Guru问题

2. 当创建一个类类型的 C++对象时，其各个部件分别是按什么顺序被初始化的？请尽量给出明确而完整的答案。解释下面这个例子中 X 对象的各个部件的初始化顺序，以便阐明你的想法。

```
class B1 { };
class V1 : public B1 { };
class D1 : virtual public V1 { };

class B2 { };
class B3 { };
class V2 : public B1, public B2 { };
```

100

```
class D2 : public B3, virtual public V2 { };

class M1 { };
class M2 { };

class X : public D1, public D2 {
  M1 m1_;
  M2 m2_;
};
```

解决方案

1. [……]下面这段代码中有什么错误？为什么？

```
// 示例14-1
//

//……

  B() : A(s = f()) {}

//……
```

别小看这行看似简单的代码，其中隐藏了一大堆相关的问题，它们都与对象生命期以及在对象存在之前使用对象这两个问题有关。注意，表达式 s=f()是传给 B 的基类 A 子对象的构造函数的实参，因而会在 A 的构造函数被调用之前执行（或者，对于本例来说，也是在 B 的任何子部件被构造起来之前执行）。

首先，这行代码试图在基类 A 子对象存在之前使用它。写这行代码的学生所用的编译器并没有指出他误用了 A::f()，之所以说误用是因为成员函数 f()是在 A 子对象上被调用的，而此时 A 子对象还没有构造起来。诚然，标准并没有要求编译器在这种情况下一定给出警告，然而你应该知道这正是标准所称的“由实现决定”的一种情况，也就是说编译器并不是一定要做，但好一点的编译器可能会做。

其次，这行代码企图在成员子对象 s 存在之前使用它，换句话说，在一个 string 成员子对象被构造之前调用它的成员操作符 operator =。

2. 当创建一个类类型的 C++对象时，其各个“部件”分别是按什么顺序被初始化的？请尽量给出明确而完整的答案。

101

下面这组规则是递归描述的：

- 首先，最上层派生类的构造函数负责调用虚基类子对象的构造函数。所有虚基类子对象会按照深度优先、从左到右的顺序进行初始化。
- 其次，直接基类子对象按照它们在类定义中声明的顺序被一一构造起来。
- 再其次，（非静态）成员子对象按照它们在类定义体中声明的顺序被一一构造起来。

- 最后，最上层派生类的构造函数体被执行。

例如，考虑如下代码。由于不管它采用的继承方式是 public、protected 还是 private，都不会影响到初始化的顺序，因此我干脆一概使用了 public 继承。

解释下面这个例子中 X 对象的各个部件的初始化顺序，以便阐明你的想法。

```
// 示例14-2
//
class B1 { };
class V1 : public B1 { };
class D1 : virtual public V1 { };

class B2 { };
class B3 { };
class V2 : public B1, public B2 { };
class D2 : virtual public V2, public B3 { };

class M1 { };
class M2 { };

class X : public D1, public D2 {
  M1 m1_;
  M2 m2_;
};
```

上例中的继承体系可以用图 14-1 表示。

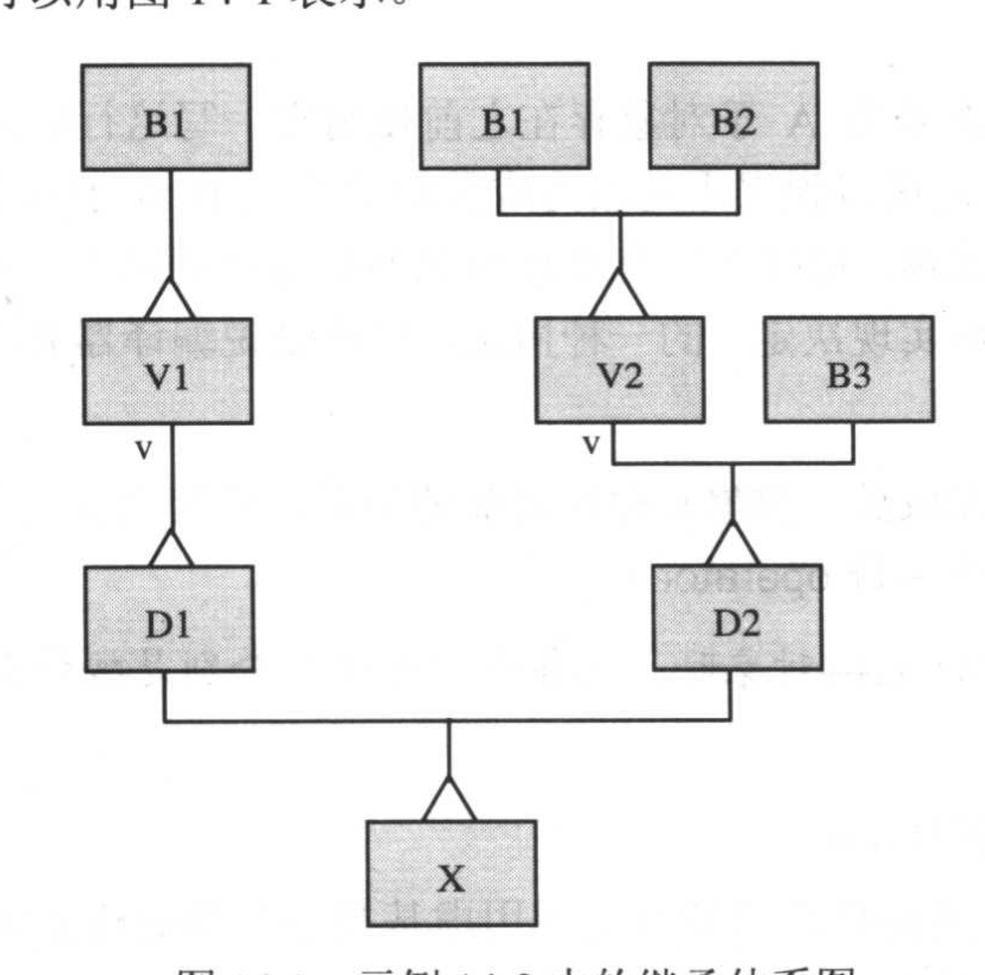


图 14-1　示例 14-2 中的继承体系图

102　在示例 14-2 中，X 对象的各子对象的构造顺序如下（以下展示的每次构造函数调用都表示其构造函数体被执行了）：

- 首先，构造虚基类子对象：

  构造V1: B1::B1() V1::V1()
  构造V2: B1::B1() B2::B2() V2::V2()

- 其次，构造非虚基类子对象：

  构造D1: D1::D1()
  构造D2: B3::B3() D2::D2()

- 再次，构造所有成员：M1::M1() M2::M2()
- 最后，构造 X 本身：X::X()

小结：关于继承的另一些话

当然，尽管该条的主要意图在于理解对象的构造顺序（以及相反的析构顺序），然而重复一下下面这个并非直接相关的准则也无伤大雅。

准则： **避免过度使用继承。**

除了友元关系之外，继承是 C++中所能表达出的最强的关系，因此若非必要请谨慎使用之。要想知道更多细节，可参考 *Exceptional C++* [Sutter00]的第 20 条，以及 *More Exceptional C++* [Sutter02]的第 19 条。

第 15 条 访问权限的使用

难度：6

谁能够真正访问类的内部？本条是关于“伪造者”、“骗子”和“偷窃者”的，另外你还将学会如何分辨并避开它们。

JG问题

1. 什么样的代码可以访问类的如下区段？

（a）public

（b）protected

（c）private

Guru问题

2. 考虑如下的头文件：

```
// 文件x.h
//
class X {
public:
  X() : private_(1) { /*...*/ }

  template<class T>
  void f(const T& t) { /*...*/ }

  int Value() { return private_; }

  // ......

private:
  int private_;
};
```

现在想要让任何调用代码都能够直接访问到该类的 private_成员，请分别给出符合以下条件的手法：

（a）一个非标准且不可移植的伎俩（hack）。

（b）一个完全符合标准且可移植的技术。

3. 这是 C++访问控制机制中的漏洞吗？如果是，也是 C++封装机制中的漏洞吗？请就此进行讨论。

104

解决方案

本条是关于“伪造者”、“骗子”和“偷窃者”的。

1. 什么样的代码可以访问类的如下区段?

简而言之:

(a) public

公用成员可以被任何代码访问。

(b) protected

保护成员可以被类自身的成员函数访问，也可以被类的友元访问，也可以被派生类的成员函数与友元访问。

(c) private

私有成员只可以被类自身的成员函数以及类的友元访问。

这便是常规的答案，而且它本身也无懈可击。然而在本条中，我们会考虑一个特例，在该特例中上面的说法不再……呃……似乎不再能够左右一切，因为 C++有时候提供了一种途径使得访问类的私有成员成为合法（如果称不上道德的话）。

2. 考虑如下的头文件: [……]

现在想要让任何调用代码都能够直接访问到该类的 private_成员，请分别给出符合以下条件的手法:

(a) 一个非标准且不可移植的伎俩;

(b) 一个完全符合标准且可移植的技术。

人们对于一些破坏性的事情（如诡秘的代码伎俩）仿佛有一种奇怪的探知欲，因此我们或许也应当由一则“事故”作为开场白:

对于非标准且不可移植的伎俩，我们能想到好几个方案。下面就是 3 个不太出名的“犯罪嫌疑人”:

嫌犯#1: 伪造者

伪造者伎俩是先将某个有待伪造的类定义复制一份，然后通过该复制后的“赝品”来达到想要达到的目的。例如:

105

```
// 示例15-1: 谎言和伪造
//
class X {
 // 这里不是#include x.h，而是手动（非法地）复制X的定义，
```

```
  // 并人为加上下面这一行:
  friend ::Hijack(X&);
};

void Hijack(X& x) {
  x.private_ = 2;            // 恶魔得逞了!
}
```

这个家伙是个伪造者。好好盯着他，因为我们可不能相信他！

当然，伪造者的所作所为是非法的，因为他违反了惟一定义规则（ODR），其中明确表示，如果一个类型（这里是 X）被多次定义，那么其所有定义都必须是完全相同的！本例中被使用的对象或许可以被称为 X 对象，而且它看起来也的确像一个 X 对象，然而它与程序其他地方使用的真正的 X 对象完全不是同一类型。

然而，这种伎俩在大多数编译器上仍然是可以工作的，因为通常真正的 X 类与伪造的 X 类的底层内存布局是一样的。如果真是这样，伪造者在汤姆·汉克斯抓到他之前倒是还能够逍遥一阵子。

嫌犯#2：偷窃者

偷窃者伎俩是指偷偷地改变类定义的含义。例如：

```
// 示例15-2: 可恶的宏伎俩
//
#define private public                    // 非法
#include "x.h"

void Hijack(X& x) {
  x.private_ = 2;                         // 恶魔得逞了!
}
```

这个家伙是个偷窃者。同样要防着他，因为他的两根手指头很灵活！

偷窃者所做的事情当然也是非法的。示例 15-2 是不可移植的，其中有两个原因：

- #define 保留字是非法的。
- 这种伎俩与伪造者伎俩一样违反了单一定义规则（ODR）。然而同样，如果类的底层内存格局没有改变，该手法或许还能苟延残喘一阵子。

106

嫌犯#3：骗子

骗子伎俩的要义在于“狸猫换太子”。如下：

```
// 示例15-3: “肮脏”的企图，企图模拟原对象的内存布局
//
class BaitAndSwitch {                     // 可能与X具有同样的内存布局
public:                                   // 从而欺骗编译器把“狸猫”当“太子”
```

```
  int notSoPrivate;
};

void f(X& x) {
  (reinterpret_cast<BaitAndSwitch&>(x)).notSoPrivate = 2;
}                                          // 恶魔得逞了！
```

这家伙是个骗子。提防着点儿，因为他会千方百计把你骗过来却卖给你价格更昂贵的假货。

毫无疑问骗子的作为也是不合法的。示例 15-3 中的代码是非法的，原因有两点：

- X 和 BaitAndSwitch 的内存布局并不保证是相同的，尽管实际上它们或许总是相同的。
- reinterpret_cast 的行为是未定义的，尽管大多数编译器都允许使用返回的结果引用，正如“骗子”所期望的。毕竟，使用 reinterpret_cast 其实就是告诉编译器相信使用者，撤消防备，不再监视使用者下面将要做的那些非法勾当。

然而，我们还是想要寻找一个完全遵从标准的、可移植的替代方案。唉，尽管许多犯罪伎俩都是低劣的、不合法的，然而也的确有些方法是遵从标准而且是可接受的。下面轮到它们粉墨登场了：

伪君子#4：语言律师

我们中许多人对那些不诚实的、整天皮笑肉不笑的律师们的反感，可能还胜过某些嫌犯的。

考虑下面的代码：

```
// 示例15-4: 合法的巧妙托词
//
namespace {
  struct Y {};
}
template<>
void X::f(const Y&) {
  private_ = 2;                            // 恶魔得逞了！
}

void Test() {
  X x;
  cout << x.Value() << endl;               // 打印出 1
  x.f(Y());
  cout << x.Value() << endl;               // 打印出 2
}
```

107

这家伙是个语言律师，他知道哪里有漏洞。他永远也不会被逮住，因为他在钻法律空子的时候会小心地遵守法律条文。留神并避开这类卑劣的家伙。

我希望可以说：“当然，语言律师所做的事情也是非法的。”然而遗憾的是，我不能，因为它实际上并不是非法的。怎么回事呢？呃……因为它利用了 X 具有一个成员模板的事实。其代码是

完全合乎标准的，标准也确保了这种代码会按照编码者的意图行事。原因是两方面的：

- 针对任何类型来特化一个成员模板是合法的。

惟一可能会带来错误的举动是将它对同样的类型作了两次特化且两次特化的方式又不同，这便违反了惟一定义规则（ODR），然而本例中并不存在这个问题，因为：

- 这里的代码使用了一个确保具有惟一性的类型，因为它位于“语言律师”自己的匿名名字空间中。因而语言确保了这种做法是合法的，而且该特化也不会影响到其他任何人进行的任何特化。

不要搞破坏

现在只剩下惟一的一个问题：

3. 这是 C++访问控制机制中的漏洞吗？如果是，也是 C++封装机制中的漏洞吗？请就此进行讨论。

该例展示了 C++中两个语言特性（即访问控制模型与模板模型）之间有趣的相互影响，我们看到，成员模板提供了一种有效的可移植的途径，绕过类的访问控制机制，从这个意义上来说似乎成员模板会隐式地“破坏封装性”。

然而实际上这并不是个问题。这里的问题在于提防“墨菲”还是提防“马基雅弗利”……，换句话说，是提防意外的**误用**（语言在这个方面已经做得相当好了），还是提防故意的**滥用**（要想做到这一点几乎是不可能的）。最后，如果一个程序员确实想要破坏类型系统，他总能找到办法的，正如示例 15-1 至示例 15-3 中所示。

108

这个问题真正的答案是：别这么干！我承认有时候一个快速的、暂时绕过访问控制机制的手段的确是蛮诱人的，例如在调试的时候能够给出更好的诊断输出……然而对于产品代码来说这可绝对不是个好习惯，你也千万别玩上瘾。而且应把这条放在你们的警告列表中。

准则： 永远不要对语言搞破坏。例如，永远不要企图通过复制类定义再添加友元声明，或提供成员模板函数特化等途径来破坏封装性。

109

第 16 条 （几乎）私有

难度：5

在C++中，在多大程度上类的private部分才算是真正私有的？本条将会看到，即便是明确定义不能由外界非友元代码访问的私有名字，也可能以某种不经意的途径泄漏给类的外部——这些不经意的途径有些是我们所熟知的，有些则不是，其中有一种途径甚至是可以精确计算并有预谋地实施的。

Guru问题

1. 快速回答：假设有 Twice()成员函数定义在另一个编译单元中，后者也参与连接，那么下面的 C++程序会正确编译运行吗？如果不能，为什么？如果能，其输出结果是什么？

```
// Twice(x) 返回 2*x
//
class Calc {
public:
  double                Twice( double d );
private:
  int                   Twice( int i );
  std::complex<float>   Twice( std::complex<float> c );
};

int main() {
  Calc c;
  return c.Twice(21);
}
```

解决方案

解决方案实际上要回答这样一个问题：在 C++中，在多大程度上类的 private 成分（如问题中的 Twice）才算是真正私有的？本条将会看到，即便是私有成员明确定义不能由非友元代码访问，然而却以某种不经意的途径泄漏给了类的外部，而这些途径有些是我们所熟知的，有些则不是，其中有两种途径甚至是可以精确计算并有预谋地实施的。 110

基本问题：可访问性

我们首先要认识到一个基本问题：与 public 和 protected 类似，private 也是一个访问控制限定符。也就是说，它也是用于指出哪些代码有权利访问它所管辖范围内的成员名字的，这就是它所能做到的全部。C++标准[C++03]中相应小节的第 11 款写道：

类的成员可以为

- private。其名字只可以被该类的成员或友元使用。
- protected。其名字只可以被该类的成员和友元，以及该类的派生类的成员和友元使用（参见[C++03]中的 class.protected）。
- public。其名字可以不受访问限制地被使用。

这些都是相当基本的东西，然而为了完整起见，让我们来看一个简单的例子，它明确显示出C++中的访问是受到良好监控的，而且确实没有任何符合标准的方式可以绕过语言的防线。示例16-1 展示了类外部的非友元代码永远不可能通过名字访问到类内部的私有成员函数，不管是直接地（显式调用）还是间接地（通过函数指针调用），因为私有成员函数的名字对于它们来说根本就是不可用的，更谈不上获取函数的地址了[1]。

```
// 示例16-1: 无法得到No::Satisfaction
//
class No {
private:
  virtual void Satisfaction() { }
};

int main() {
  No no;
  no.Satisfaction();                    // 错误!

  typedef void (No::*PMember)();
  PMember p = &No::Satisfaction;        // 错误!
  return (no.*p)();                     // 可惜……
}
```

注意，这也包含虚函数的情况。一个私有的虚函数可以被任何派生类中的相应虚函数重写，但不能被派生类访问。换句话说，派生类可以对基类中的任何同名虚函数进行重写，但不能调用（或以其他形式使用）基类中的私有虚函数的名字。例如：

111

```
// 示例16-1（续）: 派生类可以重写，但不能访问基类的私有虚函数
//
class Derived : public No {
  virtual void Satisfaction() {          // 没问题，重写
    No::Satisfaction();                  // 错误
  }
};
```

简而言之，外界（非成员或非友元）代码没有任何办法可以接触到私有函数的名字。那么前

1. 类似#define private public 这样的伎俩当然是非标准的，应受谴责，据称美国已有 42 个州立法禁止这样做。（一笑）参见第 15 条更多关于这方面的讨论以及糟糕的实践方式。

面提到的“私有到底有多私有？”这个问题现在总算有了初步的答案：

- 私有成员的名字只对其所属类的其他成员以及友元是可访问的。

这个讨论还远远没有结束（要不然本条就没什么意思了），这里涉及的问题当然不止可访问性。

另一个问题：可见性

关键字 private 确实控制了成员的可访问性。然而，与此相关的还有另外一个概念，人们通常会将它与可访问性混淆起来，这就是可见性。现在让我们回到问题中的代码：它们能通过编译且运行无误吗？

简单的答案是：不行！最初的例子中的程序代码是不合法的，无法正确通过编译。原因有两点。首先是一个相当明显的错误：

```
// 示例16-2（重复有问题的代码并加以说明）
//
// Twice(x) 返回 2*x
//
class Calc {
public:
 double                  Twice( double d );
private:
 int                     Twice( int i );
 std::complex<float>     Twice( std::complex<float> c );
};

int main() {
 Calc c;
 return c.Twice( 21 );
}
```

112

每个 C++程序员都知道，即便接受 std::complex<float>型参数的 Twice()成员函数对于 main()中的调用代码来说是不可访问的，但它仍然是可见的，且也构成了源代码依赖性的一部分。尤其是即便 main()当中的代码永远不会（也无法）关心 std::complex，永远也不能使用 Twice(complex<float>)（无法调用它，甚至不能取得它的地址），而且 std::complex 的使用也不可能以任何方式影响到 Calc 的大小或内存布局，但无论如何这段代码要想通过编译至少需要一个 complex 的前导声明。（如果 Twice(complex<float>)也是被定义为内联，那么就进一步需要 std::complex<float>的完整定义，虽然这对这段代码来说毫无意义。）

到目前为止，对于“private 在多大程度是私有的？”这个问题，我们又有了另外一些答案：

- 一个 private 成员（函数）对于任何看到其所属类定义的代码来说都是可见的。这意味着它的各个参数的类型必须事先声明，即便在当前的编译单元中永远不会使用这些参数）。

谁都知道，可以通过简单地添加#include<complex>来解决这里的问题，好吧，那我们就来试试。结果我们遇到了第二个问题，或许也是一个更隐蔽的问题：

```
// 示例16-3: 示例16-2的一个修补版本
//
#include <complex>

class Calc {
public:
  double                    Twice(double d);
private:
  int                    Twice(int i);
  std::complex<float>    Twice(std::complex<float> c);
};

int main() {
  Calc c;
  return c.Twice( 21 );          // 错误，Twice(int)是不可访问的
}
```

这个结果会令众多 C++开发者感到惊讶。有些程序员认为上面的做法是合理的，理由是对于 c.Twice(21)这个调用来说，惟一可访问的 Twice()重载版本是参数类型为 double 的，而 21 则正好可以被转型为 double，因此当然应该调用它。但事实并非如此，理由很简单：重载决议发生在可访问性检查之前。

编译器在对 c.Twice(21)这个调用进行决议时，主要做了三件事，依次如下：

113

（1）*名字查找*。在做其他任何事情之前，编译器会首先寻找一个至少包含一个名为 Twice 的实体的作用域，并将其中的候选实体列表。本例中，编译器的名字查找首先是从 Calc 的作用域中开始的，编译器会查看 Calc 中是否至少存在一个名为 Twice 的成员，如果没有，就会继续依次在其基类和外围名字空间中查找，直到找到一个至少具有一个候选函数的作用域。不过在本例中，编译器查找的第一个作用域中就具有三个候选 Twice。（要想获知更多关于 C++中的名字查找的知识，请参见 *Exceptional C++* [Sutter00]的第 31 条至第 34 条，其中讨论了 C++中的名字查找规则是如何影响打包类及其接口方式的。）

（2）*重载决议*。接下来编译器会进行重载决议，这一步的目的是在候选的重载函数中选出惟一的最佳匹配。本例中的调用参数为 21，其类型为 int，同时编译器看到，Twice()的三个可用的重载版本分别接受 double、int 和 std::complex<float>类型的参数。因此很显然，接受 int 为参数的 Twice()是最佳匹配（int 型的实参配 int 型形参属于精确匹配，无需任何转换），因此 Twice(int)被重载决议选中。

（3）*可访问性检查*。最后，编译器会进行可访问性检查，以确定被选出的函数到底是否可被调用。本例中该检查的后果当然是……完蛋了！

至于惟一可访问的函数——Twice(double)，虽说实质上也是一个可匹配的函数，然而这已经无关紧要了，它永远也不会被调用，因为这里存在着一个更好的匹配，而且匹配性的好坏永远比可访问性更重要。

有趣的是，甚至二义性匹配也比可访问性匹配更优先。考虑示例 16-3 的一个微小改版：

```
// 示例16-4(a): 引入二义性
//
#include <complex>

class Calc {
public:
  double                    Twice(double d);
private:
  unsigned                  Twice(unsigned i);
  std::complex<float>       Twice(std::complex<float> c);
};

int main() {
  Calc c;
  return c.Twice(21);             // 错误，调用具有二义性。
}
```

114

在这个例子中，永远也通不过上面提到的“决议三步曲”的第二步，即重载决议。在这一步，编译器会发现找不到一个惟一的最佳匹配，因为实参类型 int 既可以转型为 unsigned 又可以转型为 double，因而根据语言规则，这两个 Twice()被认为是同样好的匹配，正因如此，编译器无法在它们之间决出优劣，因此给出二义性调用的错误。这就是说，编译器永远也到不了第三步——可访问性检查。

甚至更为有趣的是，即便一个不可能的匹配也比一个可访问的匹配更优先。下面是将示例 16-3 重新组织了一下：

```
// 示例16-4(b): 引入“老旧”的名字隐藏
//
#include <string>

int Twice(int i);                          // 现在成了一个全局函数

class Calc {
private:
  std::string Twice(std::string s);
public:
  int Test() {
    return Twice(21);                      // 错误，Twice(string)被隐藏了！
  }
};
```

```
int main() {
  return Calc().Test();
}
```

这里同样无法通过第二步：重载决议无法在候选列表（只包含一个 Calc::Twice(string)）中找到一个可行的匹配，因为实参类型 int 无法转型为 string。因此编译器仍然执行不到可访问性检查。记住，在进行名字查找时，编译器只要查找到一个至少包含一个给定名字的作用域便不再继续查找了，即便这些所有查找到的候选函数可能都是无法调用或不可访问的。外围作用域中的其他潜在匹配则被隐藏了，永远也不会被纳入考虑。

现在，对于当初的问题："private 在多大程度上是私有的？"，又有了一些新的答案：

- **私有成员对于任何能够看到其所属类定义的代码来说都是可见的。这意味着……它会参与名字查找和重载决议，因而可能会使调用变得无效或具有二义性，即便它本身可能永远不被调用。**

115

Bjarne Stroustrup 在他的 *The Design and Evolution of C++*中曾阐述了这个语言规则背后的理念[Stroustrup94，第 55 页]：

> "如果用 public/private 控制可见性而不是可访问性，那么用户将 public 改为 private 的举动就有可能悄无声息地改变了程序的含义，使之从一个合法解释（访问 Calc::Twice(int)）变成了另一个合法解释（访问 Calc::Twice(double)）。虽说我不再认为当初的看法是结论性的（如果我曾经那样认为的话），然而当时作出的决定后来被证明是有用的，因为它允许程序员在对代码查错的时候添加或移除 public 和 private 访问指示符而同时又不会改变程序的含义。我不知道这是否算得上是一个真正的设计决策的结果。"

回到当初的问题：授权访问

关于"私有到底有多私"这个问题，答案的第一部分曾提到，私有成员只对其所属类的其他成员以及友元来说是可访问的（其名字只能被这两处使用）。注意，我故意避免了类似于"它只能被其他成员及友元调用"的说法，因为实际上后者是不确切的。可访问性是指代码使用给定名字的权利，这一点需要特别加以强调，为此我们回顾一下前面引用的 C++标准：

类的成员可以为

- ***私有的**。也就是说，其名字只可以被该类的成员或友元使用。*

如果某段代码具有直接使用某个私有成员名字的权限（本例中即指成员或友元），同时该段代码又利用该名字获取了相应函数的函数指针并将其传递给其他代码，那么接收代码就可以使用该指针，不管接收代码是否有权使用成员的名字，因为它根本不需要该成员的名字，它得到了它的指针。示例 16-5 展示了这种技巧在实际中的情况，你会看到，一个具有对 Twice(int)名字的访问权的成员函数利用该职权将 Twice(int)的指针泄漏出去：

```
// 示例16-5: 授权访问
//
class Calc;
typedef int (Calc::*PMember)(int);

class Calc {
public:
  PMember CoughItUp() { return &Calc::Twice; }

private:
  int Twice(int i);
};
int main() {
  Calc c;
  PMember p = c.CoughItUp();        // 对Twice(int)的访问
  return (c.*p)(21);                // ok
}
```

116

也可以参见[Newkirk97]，其中详细描述了实现这种故意泄漏私有的手法的一种有用应用。

那么，对于当初的问题，答案又进了一步：

- **具有对某个成员的访问权的代码可以通过泄漏该成员的指针（摆脱了名字的束缚）的方式将其访问权限授予其他任何代码。**

最后，这里还有另一种做法，使用该做法的类可以向外界提供一种具有完美可移植性且符合标准的途径来访问它们的私有成员，该做法就是成员模板。在第 15 条曾经讨论过访问某个类的私有成员的若干途径。其中绝大部分都是非法的或是在标准的法律之外操作的，然而其中有一条途径却是完全符合标准的：

```
// 示例16-6: 采纳示例15-4中的做法
//

// 某个头文件中
//
class X {
public:
  template<class T>
  void f(const T& t) {
    // ......
  }

// ......

private:
  int private_;
};
```

```
// 使用代码
//
namespace {
 struct Y {};
}

template<>
void X::f(const Y&) {
 private_ = 2;                    // 得逞了！
}
```

117

这儿发生了什么？简而言之，任何成员模板都可以针对任何类型进行特化。因此可以使用一个特殊的类型来特化它，前提是其他任何人都不会使用你所使用的这个类型来特化该成员模板，例如可以使用一个定义在自己的匿名名字空间中的类型来作为特化的模板实参，于是……猜猜发生了什么？你为所特化的成员模板所属的类写了一个成员函数！而成员函数具有对其所属类的所有成员的访问权。

因此，对于本条开始提出的“private 到底在多大程度上是私有的?”这个问题，我们终于找到了答案的最后一部分：

- **private 成员的名字只对其所属类的其他成员或友元来说是可访问的，而这里的其他成员也包括成员模板的任何显式特化（不管它的某个给定的显式特化是否为意料之中的）。**

小结

那么，private 到底在多大程度上是私有的呢？下面就是我们得到的答案：

private 成员的名字只对其所属类的其他成员或友元来说是可访问的，而这里的其他成员也包括成员模板的任何显式特化（不管它的某个给定的显式特化是否为意料之中的）。不过，如果某段代码具有对某个成员的访问权，该段代码就可以通过泄漏该成员的指针的方式将其访问权限授予外界的任何代码。

private 成员对于任何能够看到其所属类定义的代码来说都是可见的。这意味着它的函数参数的类型必须事先声明（即便它们可能对于当前编译单元来说是无用的），另外它还会参与名字查找和重载决议，因而可能会使调用变得无效或具有二义性（即便它本身可能永远不被调用）。

118

第 17 条　封　　装

难度：4

“封装”对于C++程序员来说究竟意味着什么？恰当的封装和访问控制对于成员数据来说意味着什么——它们应当总是公用的或保护的吗？本条主要讨论该问题的替代方案，并说明这些方案对你的代码的健壮性会产生怎样的影响。

JG问题

1. “封装”意味着什么？它对于面向对象设计和编程的重要性如何？

Guru问题

2. 类的非静态数据成员在哪些情况下（如果存在的话）应当是 public、protected 还是 private 的？将你的答案归结为一条编码准则。

3. std::pair 类模板使用了公用的数据成员，因为它并非一个封装类，而只是一个捆绑数据的机制。设想这么一个类模板：它类似于 std::pair，只不过额外提供了一个 deleted_标志，后者可以设置、查询，但不可以处于未设置状态。显然，该标志本身必须为私有的，因为只有这样才能防止用户直接将其置于未设置状态。如果我们选择让其他数据成员成为公用的，就像它们在 std::pair 中那样，那么就会得到如下的类：

```
template<class T, class U>
class Couple {
public:
  // 主要数据成员是公用的……
  T first;
  U second;

  // ……但仍然存在着类机制以及私有成员。
  Couple()                    : deleted_(false) {}
  void MarkDeleted()          { deleted_ = true; }
  bool IsDeleted()            { return deleted_; }

private:
  bool deleted_;
};
```

在以上代码中，其他数据成员（即 first 和 second）应当仍为公用的吗？为什么或为什么不？如果答案是肯定的，那么这个例子是说明在同一个类中混合公用成员和私有成员的做法也可能是良好设计的有力论据吗？

119

解决方案

1. “封装”意味着什么？

封装（encapsulate）一词在字典中是这样定义的：

en-cap-su-late [动词] 用囊状物包围、包装或者保护。

而在程序设计当中，封装恰好具有相同的意思：即通过将类的内部实现隐藏在一层外围的、外界可见的接口后加以保护。

同时，囊状物（capsule）一词的定义进而又为我们设计一个良好的类接口指明了方向：

cap-sule [名词]

1a:一层膜状或囊状物，包裹着器官或组织……
2:包含着孢子或种子的一个封闭的腔……
4a:包装有药物的胶囊……
5:金属的密封件……
6:……包围某些微小有机体的包膜……
9:飞行员或宇航员所处的一个狭小的加压的密封舱……

注意在上面的解释中重复出现的几个字眼：

- 包围、包装、包含、包裹

一个良好的类接口能够隐藏其所属类的内部实现，并向外界提供一个与类的内部实现分离和区别开来的“界面”。由于囊状物仅用于包裹一组内聚的（cohesive）子对象，因而其接口也应当是内聚的，换句话说，构成该接口的各成分互相之间应当是直接相关的。

细菌细胞膜外表面的意义在于与外界进行感知、接触以及发生交互作用。（若是对内部而言这些意义就没那么关键了。）

- 封闭的、密封

一个良好的类接口设计必须是完整的，同时又不能暴露类的任何内部实现。接口的作用只是作为一层密封层，而且通常是作为代码防火墙（编译时或运行时的，或既是编译时又是运行时的），令外界代码无法依赖于类的内部实现，从而确保类的内部实现的更动不会影响到外界的代码。

如果一个细菌的细胞膜不是封闭的，那么这种细菌肯定活不长；其内部的有机组织会迅速溢出，它也肯定会死亡。

120

- 保护、外壳

一个良好的类接口会保护类的内部实现，以免它们遭受到未经授权的访问和操作。特别地，类接口的主要作用在于确保所有对类的内部结构的访问和操纵都不会破坏类的不变式。

杀灭细菌（和人类）的主要手段是制造能够破坏细胞膜的东西。从微观层面来说，这包括化

学制品、酶或者能够适当制造破孔的有机生物体（或许最终会是纳米机器人）。而从宏观层面来说，刀和枪都是人类长久以来热衷的武器。

封装在面向对象中的地位

封装对于面向对象设计和编程的重要性如何？

封装是面向对象编程的核心概念。

其他面向对象技术，如数据隐藏、继承和多态，从原则上来说也是重要的，因为它们支持封装的某些特殊形式。例如：

- 封装几乎总是意味着数据隐藏。
- 运行时多态，使用虚函数，更彻底地将内部实现（由派生类提供）与对外接口（由基类提供）分离，提供内部实现的派生类甚至可以在最终用到它的客户代码之后编写。
- 编译时多态，使用模板，将接口与实现完全分离，任何具有必备操作的类都可以替换使用，而无需具有任何继承或其他关系。

封装并不总是数据隐藏，不过数据隐藏总是某种形式的封装。封装并不总是多态，而多态则总是某种形式的封装。

所谓面向对象，通常是这么定义的：

数据以及操作这些数据的函数的组合

以上定义从某种程度上来说是正确的，只不过它忽略了那些虽为非成员函数，但从逻辑上来讲仍是类的接口的一部分的自由函数，例如 C++中的 operator<<。同时该定义也强调了高内聚性。不过，该定义并没有充分强调面向对象的另一个重要方面，即：

通过由一组函数构成的接口将调用代码与内部数据隔离开来（由接口来负责操纵内部数据） 121

这一方面强调了两点，一点是低耦合度，另一点是被组合起来的函数构成了一个起保护作用的接口。

简单地说，面向对象就是关于将接口与实现分离的方法学，只不过这种分离必须能够改善高内聚性和低耦合度，这两者远在“对象”被发明之前就已公认为良好的软件工程目标了。这些概念解决了依赖性管理的问题，后者是现代软件工程中的一个关键概念，特别是对于大型系统来说。（参见[Martin95]及 Martin 在 1996 年发表的几篇文章，尤其是题目中有“principle”的那几篇，发表于[ObjectMentor]。）

Public，Protected还是Private？

2. 类的非静态数据成员在哪些情况下（如果存在的话）应当是 public、protected 还是 private 的？将你的答案归结为一条编码原则。

通常我们会首先看一下规则是什么，然后看看它有哪些例外。不过这次让我们采取另一种做法，先来看看例外情况。

对于下面列出的一般规则来说，惟一的例外是所有的类成员（不管是函数还是数据）皆为公用（就像 C 风格的 struct 一样）的时候。这种情况下我们所谓的“类”其实并非功能完整的类，功能完整的类至少得有自己的接口、行为以及不变式，而我们的所有成员皆公用的“类”甚至连半个类都算不上，只能算是一堆数据（bundle-o-data）。我们的“类”仅仅是为方便起见将一组对象捆绑在一起，这没什么问题，尤其是这种做法能够向后兼容那些操纵 C 风格 struct 的 C 程序。

然而，除了上述特殊情况之外，数据成员永远都应当为私有的。

公用数据违反了封装原则，因为它允许调用代码直接操纵对象的内部信息。这意味着对调用代码的高度信任！毕竟在现实当中，很少有人需要直接去操纵对象的内部实现（举个例子，很少有人需要直接操纵我的胃），因为那样他们可能很容易一不小心就干错事。至多他们只会通过我的公用接口去间接地操纵我的内部，这种做法的好处是他们在进行操作之前必须首先让我知道并得到我的认可才行（例如，递给我一瓶标有“喝我”字样的液体，那么我可以决定是喝了它，还是用它来洗头或是洗车，根据自己的感觉和判断）。当然，有些人的确有资格直接操作我的“内部”（例如外科医生），只不过即便如此：（a）这种情况也很少见；（b）我至少能够选择是否愿意接受手术；（c）我至少能够选择我高度信任的医生。

类似地，大多数调用代码永远都不应该直接去操纵类的内部（例如，通过查看或改变类的成员数据），因为它们可能很容易一不小心就干了错事；至多它们只能通过被操纵类的公用接口，在让被操纵类获知并得到其认可的前提下间接地操纵其内部（例如将一个 Bottle("Drink Me")对象送给另一个对象的一个公用成员函数，至于到底用不用这个 Bottle("Drink Me")对象或者用它来做些什么事情，那就是后者的事了）。当然，有些非成员代码可能的确有资格直接访问某个类的内部（通常这些代码应当是成员函数，不过也有个别例外，譬如 operator<<一般就是非成员函数），然而即便如此：（a）这种情况也属罕见；（b）被操纵类有权选择是否声明其为 friend；（c）被操纵类有权选择将这类外界代码中的哪些声明为具有一定信任级别的 friend。

122

简而言之，公用数据是危险的（除非是对于 C 风格 struct 而言）。

类似地，保护数据也是危险的（这次没有例外）。

“等等，”或许有人会说：“第一点我倒是同意的，不过为什么说保护数据也是危险的呢？”这是因为刚才我们的论点同样可以用在保护数据身上：保护数据也是接口的一部分——保护接口也是一个外界可见的接口，只不过这里的“外界”只是公用接口“外界”的子集而已，因为保护接口的外界就是指派生类中的代码。但是为什么说这又是没有例外的呢？原因是保护数据决不仅仅是一堆数据；如果它真是的话，那也只有其派生类能够将它当成一堆数据，况且你又何曾将某个基类的额外实例只是作为一堆数据使用过呢？那样做无疑很怪异。

要想知道为什么当初 C++语言要加入保护数据的历史原因，以及为什么当初支持它的人现在

也同意这是个糟糕的主意，可以参考 Stroustrup 的 *The Design and Evolution of C++* [Stroustrup94]。

准则：总是将所有数据成员放在私有区段。惟一的例外是C风格的struct，后者的意图并不在于封装什么东西，因而其所有成员都可以是公用的。

一个通用的形式转换

下面让我们来证明“成员数据应当总是私有的”这一原则。我们可以采取反证法：先假设公用/保护的成员数据也是恰当的，然后指出在这一假设之下数据成员都不应该为公用/保护的。

```
// 示例17-2(a): “危险的”非私有数据
//
class X {
 // ……
public:
 T1 t1_;
protected:
 T2 t2_;
};
```

首先我们注意到，以上的类总是可以被转换成其他等价形式，同时又不会丧失一般性或效率： 123

```
// 示例17-2(b): 封装的数据（良好）
//
class X {
 // ……
public:
 T1& UseT1() { return t1_; }
protected:
 T2& UseT2() { return t2_; }
private:
 T1 t1_;
 T2 t2_;
};
```

这就是说即便有理由允许对 t1_或 t2_的直接访问，我们也有上面这个简单的等价形式，它能够通过一个（最初为内联的）成员函数来实现对成员的访问。示例 17-2(a)和 17-2(b)是等价的。但前者是否更好一些呢？

为了证明示例 17-2(a)中的做法决不该被采用，我们只需说明：

（1）示例 17-2(a)的做法并不具有 17-2(b)的做法所不具备的优点；

（2）示例 17-2(b)的做法具有一些实实在在的优点；

（3）示例 17-2(b)的做法并不需要任何代价。

倒过来看上面这几点：

第 3 点是不言而喻的。UseT1()这样的内联函数是按引用返回的，因而不会带来任何复制开销，所以它很可能会被编译器给完全优化掉。

第 2 点也显而易见：只需考虑一下源代码依赖性就明白了。在示例 17-2(a)中，所有操纵 t1_和/或 t2_的调用代码都是直接通过名字来引用它们的；而在示例 17-2(b)当中则是通过 UseT1()/UseT2()这类函数的名字来操纵它们。示例 17-2(a)更为苛刻，因为 t1_或 t2_的任何改动（例如将它们删除并以其他成员来取代它们，或仅仅是添加一些其他功能）都会导致所有用到它的代码不得不进行相应的变动。而在 17-2(b)当中，我们可以在不影响调用代码的前提下随意地添加功能，而且甚至就连 t1_和 t2_都可以被完全删除，因为成员函数才是负责构成类的接口的部件，也是“包围”、“密封”以及“保护”类内部实现的一层保护膜。

124

最后，我们说示例 17-2(a)中调用代码能够直接对 t1_和 t2_做的事情通过示例 17-2(b)中的间接调用 UseT1()和 UseT2()也能够完成，因而第 1 点也得到了保证。只不过在后一种情况当中调用者可能会需要写一对额外的圆括号而已。

下面让我们来考虑一个具体的例子：假设你想要将某些功能，譬如说计算对 t1_和 t2_的访问次数这一功能添加进类当中。如果该功能需要一个数据成员来记录访问次数（像示例 17-2(a)中那样），那么下面的一切就是无可避免的了：

（1）你得创建访问函数来实现相应的功能，并同时将相应的数据成员设为私有的。（换句话说，还是走到了示例 17-2(b)的路上，只不过是在后期作为一种改进而已。）

（2）类的所有使用者可有苦头吃了：他们得将代码中任何用到 t1_或 t2_的地方全都改动一番，改成功能上无甚区别的另一种形式（调用访问函数）。这对于那些已经被最后期限逼得焦头烂额，手头还有一大堆工作要做的开发者来说可真是火上浇油！这下你的用户可得感激你了，记住，当他们买来礼物送给你时，如果你听到包装盒里面有滴答声，千万别打开！

（3）所有客户代码都得重新编译。

（4）如果漏改了某些地方，会导致编译不通过，结果又得返回重复步骤 2 和 3，直到所有地方都修改完毕。

然而如果你原本就已经有了简单的成员访问函数（像示例 17-2(b)中那样），那么只需做下面这两件事情：

（1）你在访问函数内部作一些相应的改动。

（2）所有的客户代码一般只需重新连接即可（如果你所改动的函数是位于一个分离编译的.cpp 文件当中，不是内联函数），或者，最坏的情况，客户代码都需要重新编译（如果改动的函数位于头文件之中）。

而在实际当中，最糟糕的问题是，如果你一开始采用的就是示例 17-2(a)中的做法，那么到

了后期你可能根本就不被允许对其作出上述的改动（即改为示例 17-2(b)那样的形式）。依赖于一个接口的用户越多，要想改变该接口就越是困难。这就将我们带到了一个准则面前。该准则与第二定律是等价的。

准则：接口是最需要在第一时间做对的事情。其他东西都可以在后期进行修正。如果你一开始就没有把接口做对的话，那么以后你可能就永远没有机会去改正它了。

一旦一个接口得到了普遍的使用，那么许多人就会因此而依赖于它，这时再去对它进行改动也就变得不切实际了。诚然，接口总是可以被扩充的（“添加”，而非“改动”），这种扩充并不会影响任何客户代码，然而仅仅是添加成员函数并不能帮你解决你如今认为是糟糕的设计，它至多能让你添加几个替代做法，后者则往往会让用户们感到无所适从，他们理所当然地会问：“为什么完成同一件事情会有两种（三种，或 *N* 种）途径？我到底该选哪一种？” 125

简而言之，一个糟糕的接口设计在木已成舟之后就是很难或者无法更改的了。所以你应当尽量在第一时间就将接口做对，并让它发挥包裹、密封以及保护类的内部实现的作用。

佐证

3. std::pair 类模板使用公用数据成员是因为它并非一个封装类，而只是一种捆绑数据的机制。

注意，这是公用数据的一种完全合理的使用方式。不过即便如此，std::pair 如果使用访问函数的话也肯定不会比现在差。

设想这么一个类模板：它类似于 std::pair，只不过额外提供了一个 deleted_标志，后者可以设置、查询，但不可以处于未设置状态。显然，该标志本身必须为私有的，因为只有这样才能防止用户直接将其置于未设置状态。如果我们选择让其他数据成员成为公用的，就像它们在 std::pair 中那样，那么就会得到如下的类：

```
// 示例17-3(a): 混用公用数据和私有数据？
//
template<class T, class U>
class Couple {
public:
  // 主要数据成员是公用的……
  T first;
  U second;

  // ……但仍然存在着类机制以及私有实现。
  Couple()                    : deleted_(false) {}
  void MarkDeleted()          { deleted_ = true; }
  bool IsDeleted()            { return deleted_; }
private:
```

```
    bool deleted_;
};
```

在以上代码中，其他数据成员（即 first 和 second）应当仍为公用的吗？为什么或为什么不？如果答案是肯定的，那么这个例子是说明在同一个类中混合公用成员和私有成员的做法也可能是良好设计的有力论据吗？

这个名为 Couple 的类是违反通常编码准则的一个反例。它试图展示一个“差不多是 struct”的类，只不过该类具有一些私有的普通数据而已。这些普通数据（本例中只是一个简单的属性）具有一个不变式。其设计者声称这个普通数据属性与 Couple 内含值的更新完全无关。

126

让我们从“完全无关”这种说法开始：首先我并不相信这种说法。标志变量的更新动作可能是独立的，然而它所对应的属性显然不可能独立于 Couple 的值，否则它就不必与 Couple 的其他数据成员内聚地组合在一起了。显而易见，deleted_属性并不独立于它所从属的对象，而是“作用于”的关系。

请注意，我们可以不再使用公用与私有混用的办法，可以借助于访问函数来实现我们的解决方案，即便访问函数一开始可能得返回引用：

```
// 示例17-3(b): 恰当的封装，一开始可以使用内联的访问函数。日后如果有必要的话，
// 这些函数也许会“成长”为非平凡的非内联函数；如无必要则可保持现在的形式。
//
template<class T, class U>
class Couple {
  Couple()                : deleted_(false) { }
  T& First()              { return first_; }
  U& Second()             { return second_; }
  void MarkDeleted() { deleted_ = true; }
  bool IsDeleted()     { return deleted_; }

private:
  T first_;
  U second_;
  bool deleted_;
};
```

“啊哈？”人们可能会立即发出疑问：“为什么非得写几个什么也不干的访问函数呢？”答案是：正如我们前面在讨论示例17-2(b)的时候提到的，如果今天调用代码可以改变该对象的某些方面（本例中就是指deleted_属性），那么明天你或许就会想要添加某些新特性（即便它们可能无非是添加一些调试信息或添加一些检查而已）。示例17-3(b)允许你这么做，而且这种灵活性并不会给你带来任何效率上的开销，因为函数是内联的。

例如，假设未来的某个早晨你决定要对所有对标记为“删除”的对象的访问进行检查，那么：

- 如果先前采取的是示例 17-3(a)的做法，现在就不能这么做了。至少在不改变接口设计，

也不要求所有使用了 first_和 second_的代码重编写的前提下不能。

- 而如果先前采取了示例 17-3(b)中的做法的话，现在只需简单地将负责检查的代码塞入 First 和 Second 这两个成员函数之中即可。这里的改动对于之前和之后 Couple 类的使用者来说都是透明的。客户代码至多需要重新编译一下，根本无需进行任何源代码上的改动。

127

此外，似乎示例 17-3(b)的做法在现实中还有其他一些实际的附带好处。例如，正如 Nick Mein 指出的："你可以在访问函数中设置一个断点（或其他什么东西），以便测出何时何地你的值正在被改动。这在跟踪调试的时候会是相当有用的功能。"这毫无疑问，而且通常的确如此。

小结

除了 C 风格的 struct 这一特殊情况（其中所有成员皆为公用的）之外，所有数据成员都应当总是私有的。如果不这么做就会违反本条开始指出的所有封装原则，并使客户代码与成员的名字耦合起来，后一点将使以后要将这些成员正确封装更加困难（几乎不可能）。永远没有什么充分的理由支持你去写公用的或保护的数据成员。它们总能够被轻易地包装在内联的访问函数之后，这不会带来任何开销，因此在第一时间就将事情做对总是正确的。（诚然，标准库中的确有些类使用了保护的数据成员，不过这些例子属于"切勿仿效"者之列。）

在第一时间就将接口设计正确。内部实现在事后皆可以容易地加以改正，但一旦将接口搞错了的话，你可能就永远也没机会去改正它了。

128

第18条 虚 拟

难度：7

本条中，我们将会深入讨论一些老问题，不过我们会给出新的、改进的解决方案。对于有关虚函数的两个老问题，最新的答案直接引出了4个编码准则，它们会带来更为健壮的类设计。这些编码准则解答了下列问题：为什么接口应当为非虚的？为什么“虚”的东西应当为私有的？以及古老而陈旧的关于基类析构函数的建议存在什么问题？它们真地是正确的吗？

JG问题

1. 关于基类析构函数的“一般建议”是什么？

Guru问题

2. 虚函数什么时候应当为 public，什么时候应当为 protected，什么时候又应当为 private？证明你的答案，并举出已有的做法。

解决方案

本条我将会给出有关虚函数的两个老问题的最新答案。这些答案接着会直接引出 4 则有关类设计的准则。

我们要回答的问题其实已经是老问题了，然而人们总是会一再提问，而且其答案中有些内容也随着时间的推移以及我们在现代 C++中获得的经验积累而一再变更。

关于基类析构函数的一般建议

关于基类析构函数的“一般建议”是什么？

我知道你曾听过这个问题：“基类的析构函数应当为虚函数吗？”

唉。我多希望这仅仅是个常常有人问起的问题。然而遗憾的是，它其实是一个经常被人们争论的问题。

这个问题通常的答案是：“哈？基类的析构函数当然应该是虚的！”只可惜这个答案是错误的，而且 C++标准库本身当中就包含了这方面的例证，只不过这个答案经常看起来很正确，足以给人一种假相。

129

待会我们再回到这一问题，这只是有关虚函数可访问性的两个问题中的一个。让我们从其中更具一般性的问题开始。

虚函数问题之一：公用还是私有？

我们需要考虑的一般性问题是：

虚函数什么时候应当为 public，什么时候应当为 protected，什么时候又应当为 private？证明你的答案，并举出已有的做法。

简单的答案分别为：很罕见（如果有的话）、有时、默认情况下。前面我们已经看到，对于其他类成员来说答案也是一样的。

我们中的大多数人都曾在一些痛苦的经验当中学到要将类的所有成员默认放在私有区段，只在绝对有必要将它们暴露给外界的时候才将它们放到公用区段。这当然是良好的封装准则。我们在之前也了解到，数据成员始终应当为私有的（除非是对于 C 风格的 struct 而言，后者只不过是一组数据的简单捆绑，其意图并非要去封装什么，见第 17 条）。相同的做法同样可以用到成员函数身上，因此我建议遵循如下的准则，它可以被总结为一句对设置为私有之优点的陈述（至少对于代码来说是有好处的）。

准则：尽量让接口成为非虚的。

注意，我说的是“尽量”，而不是“总是”。

有趣的是，C++标准库中的类极大程度上都遵循了该准则。如果撇开析构函数不算（我们将在本条的第 4 个准则中单独讨论它），另外那些在类模板的特化中再次出现的相同虚函数也不重复计入的话，标准库中算来一共有：

- 6 个公用的虚函数，皆为 std::exception::what 及其重写版本；
- 142 个非公用的虚函数。

最近我遇到一个重要的例子。在写作本书的同时，我还在从事微软的.NET 平台以及.NET 框架（FX）的 C++方面的工作，后者正在逐渐演变为 WinFX，WinFX 是 Win32 API 的面向对象新版本（下一代 Windows API），也是微软的下一代操作系统 Longhorn 的编程模型，该操作系统大约在 2005、2006 年内发布[1]。而 WinFX，即便是目前仍在开发的内部形式，也已经是一个庞大的 API 了：到我写作本书为止，WinFX 已经包含了超过 14000 个类以及近 100000 个成员函数（包括所有当前已发布的.NET 框架的内容以及更多辛苦开发的新成果）。是的，WinFX 非常庞大。 130

为什么要提到.NET 框架以及它向 WinFX 的演化呢？因为对于一个如此庞大的类库来说，人们会希望存在一些良好的类设计准则，这些准则会得到严格的遵守。现在我可以愉快地宣布，就这两者来说你们都是对的。以下就是 WinFX 的设计准则之一，你看到之后可能会觉得太熟悉了，尽管我也同意它，但我并没有对它施加任何影响……它完全是由于一些类似的、有所共通的原因而裁定的：

1. Longhorn 的正式名称已经定为 Windows Vista。——编者注

我们建议通过保护的（家族）方法来提供定制能力。基类的公用接口应当提供一组丰富的功能以供其用户使用。然而，该类的定制者通常希望尽可能通过实现最少的方法就能向用户提供这组丰富的功能。为了达到这个目的，你应当提供一组非虚的或 final 的公用方法并让它们去调用一个单一的、具有“Core”后缀的、提供了具体实现的保护（家族）方法。该做法在设计模式领域也被称为“模板方法”（Template Method）模式。——摘自《.NET 框架（WinFX）设计规范》，内部草案，2004 年 1 月[1]

为什么说这一在实践当中反复出现的模式是个好点子呢？让我们来看看。

传统上，许多程序员习惯于编写具有公用虚函数的基类。例如，我们可能会写：

```
// 示例18-1: 一个传统的基类。
//
class Widget {
public:
  // 这些函数当中每个都可被选作纯虚函数，如果这样，
  // 它们在Widget中有可能实现，也可能没有实现，
// 参见More Exceptional C++[Sutter02]23。
  //
  //
  virtual int Process(Gadget&);
  virtual bool IsDone();

  // ……
};
```

这些公用的虚函数，正如所用公用的虚函数一样，同时既给出了接口又给出了可定制的行为。问题在于“同时”上，由于每个公用虚函数都得为两个具有不同需求和目的的客方服务：

131

- 其一是类的外部调用者，它们依赖于其公用接口来使用它。
- 另一个是派生类，它们同样依赖于虚函数的“定制接口”，因为通过这些接口它们才能扩展和定制它们的基类。

一个公用虚函数被强制做两件事情：其一是给出接口，由于它是公用的，因此就成为了 Widget 呈现给外部世界的接口的直接的一部分。其二是给出实现细节，即其内部可定制的行为，由于它是虚的，因而就提供给了派生类一个替换其基类中的实现（如果有的话）的机会。一个公用的虚函数天生就会做两件截然不同的事情，它们的客方相互之间是一种竞争的关系，这意味着公用虚函数没能很好地将概念区分开来；我们应当考虑采用其他途径。

如果我们想要将“给出接口”与“给出实现的可定制行为”分离开来的话，我们就会发现最终的结果会让我们立即联想到模板方法模式[Gamma95]，因为它们非常类似。然而，这里的情况又有所不同，与模板方法相比，这儿采用的方法的目的要狭窄得多，因而我们给它一个更为专门

1. 根据此规范出版的《.NET 框架设计规范》一书即将由人民邮电出版社出版。——编者注

的名字：非虚接口（Nonvirtual Interface, NVI）模式。下面就是它在实际运用中的情形：

```
// 示例18-2: 一个更“摩登”的基类，使用了非虚接口模式
// 来将接口与内部实现分离开来。
//
class Widget {
public:
 // 稳定的、非虚的接口
 //
 int Process(Gadget&); // 使用DoProcess...()
 bool IsDone(); // 使用DoIsDone()

 // ……
private:
 // 定制属于实现细节，它可能与接口直接相关，也可能无直接关系。
 // 下面的这些函数皆可为纯虚函数，
 // 如果为纯虚函数则在Widget类中可能实现，也可能没有实现，
 // 参见《More Exceptional C++》[Sutter02]中的第23条。
 //
 virtual int DoProcessPhase1(Gadget&);
 virtual int DoProcessPhase2(Gadget&);
 virtual bool DoIsDone();

 // ……
};
```

132

尽量使用非虚接口模式来使接口稳定和非虚，将可定制的工作放在那些负责实现可定制行为的非公用虚函数当中。毕竟，虚函数是被设计用来允许派生类定制行为的，所以最好别让公用派生类也将继承而得的接口给定制了，后者理应是一致的。

非虚接口的做法具有如下几个优点，同时并不存在显著的缺点：

第一，注意到基类现在对它的接口及策略拥有了完全的控制权，因而可以很方便地在一个单一的、可复用的地方实施（enforce）接口的前条件跟后条件、插入一些设备（instrumentation），或者做一些类似的事情等等，这个“单一的、可复用的地方”就是非虚接口函数。这促进了良好的类设计，因为它允许基类对派生类的定制（替换）举动实施一致性检查（根据 Liskov 替换原则[Liskov88]），只要基类认为有意义，就完全可以进行任何程度的检查。如果需要考虑效率的话，基类可以选择只在调试模式下进行一些特定的前条件和后条件的检验，例如，我们可以通过在构建项目时选择非调试模式（release）从而阻止编译器编译所有的检查代码，或者我们也可以通过一个可配置的调试模式来允许在运行期禁用选定的检查代码。

第二，由于我们更好地分离开了接口与实现，所以我们可以让它们分别具有自然而然的形式，而不是费力寻找一个强迫它们看起来一模一样的折衷形式。例如，注意到在示例 18-2 中我们发现让我们的用户看到一个单一的 Process 函数，同时又允许用户在两个部分（DoProcessPhase1

以及 DoProcessPhase2）单独作更为灵活的定制的做法更有意义。如果使用公用虚函数的话，不将这种概念上的分离显式地呈现在接口上是不可能达到同样效果的，但这么一来就会增加复杂度，用户得知道更多信息才能够在两个函数之间进行正确的调用。（参见 *Exceptional C++* [Sutter00] 条款 19 中的一个相关例子的更多讨论。）

第三，采用非虚接口模式可以让基类在变化面前更为稳固。我们大可以在基类完成之后再改变主意，如添加前条件及后条件检查，或者将某件事分割成更多步来完成，或者进行重构，或者使用 Pimpl 惯用手法（参见 *Exceptional C++* [Sutter00]）来实现更为完善的接口/实现分离，或者对 Widget 的可定制化能力进行一些其他的调整，而所有这些都可以在毫不影响 Widget 的客户代码的前提下完成。例如，要想一开始使用公用虚函数并在以后试图对其添加前条件和后条件检查的话会困难得多，比预先提供一个非虚的外覆转发函数（即便当时并不立即需要实施检查或其他额外工作）并在以后需要的时候再向其中插入检查代码的做法要更为困难。（你可以在[Hyslop00]中看到关于像 Widget 这样的类如何才能变得更为稳固，更能适应以后的版本更动及重构的讨论。）

133

“但是但是但是……”有人可能会指出：“假设所有的公用非虚函数一开始所做的就是将调用转发至对应的私有虚函数，那么这也仅仅是一行短短的、愚蠢的代码而已，是不是相当无用呢，而且真的什么坏处也没有吗？我们是不是损失了一些效率（额外的函数调用）、增加了一些复杂性（额外的函数）？”答案皆是否定的。首先，关于效率：在实际当中这种做法丝毫无损效率，因为如果公用非虚函数是只包含惟一一行转发语句的**内联函数**的话，那么我所知道的所有编译器都能够完全将它优化掉，不会留下任何额外开销[1]。其次，关于复杂性：惟一的复杂性是我们需要消耗一点额外的时间来编写只有一行的通道函数，这只是小事一桩。以上是最“C”式的吹捧。另外，接口完全不受影响：对类的外界用户来说，我们的类仍然具有跟原来一样数目的公用函数，而对使用它的派生类的程序员来说，它仍然具有跟原来一样多的虚函数。不管是呈现给外界的公用接口还是呈现给派生类的继承接口都丝毫不会变得更为复杂。这两种接口被明确地区分了开来，这是好事情！

好了，这番陈述应当证明了非虚接口的正当性，并告诉我们将虚函数置为非公用能够带来好处，不过我们还没有回答虚函数到底该为私有的，还是该为保护的。因此让我们继续：

准则：尽量将虚函数置为私有的。

这很简单。它允许派生类根据需要通过重写函数来定制某些行为，同时又不会将这些虚函数暴露给派生类（如果将它们置为保护的，派生类就可以直接调用它们）。关键是虚函数的存在是为了允许定制；除非它们也需要被派生类中的代码直接调用，否则根本不需要将它们设为非私有的。不过话说回来，有些时候我们的确需要调用基类中的虚函数（参见[Hyslop00]中的例子），只有当出现这种情况的时候才有必要将虚函数设为保护的。因此我们有如下的准则：

---

1. 实际上，有些编译器总是会将这种函数置为**内联**并将它优化掉，而不管你个人实际想不想这样，不过这又是另外一个问题了；见第 25 条。

准则：只有当派生类需要调用基类中实现的虚函数的时候，我们才需要将后者设为保护的。

至少，非虚接口模式运用到虚函数身上能够很好地帮助我们将接口与实现分离开来。当然，将这种分离变得更彻底也是可能的，只需借助于一些途径彻底将接口跟实现隔离开来即可，例如桥接（Bridge）[Gamma95]这类模式或者 Pimpl 这类惯用法（该手法主要用在控制编译期依赖性和异常安全保证方面）[Sutter00,Sutter02]，或者更为一般的 handle/body 或 envelope/letter[Coplien92]惯用法，等等。除非你需要更为彻底的接口/实现分离，否则非虚接口模式通常足以满足你的需求。另一方面，我认为非虚接口模式的这种运用也是一个应当被默认采用的良好做法，同时我们也应当将它看作新编写的代码至少应当具备的必要的分离程度。毕竟，这种做法不会带来任何开销（除了需要额外写一行代码之外），却能够为以后省却许多烦恼。 134

请参见[Hyslop00]中关于使用非虚接口模式来私有化虚（多态）行为的例子。

说到[Hyslop00]，你有没有注意到这儿的代码中有一个公用虚析构函数？这将我们带入了下一个主题：

虚函数问题之二：基类的析构函数——又怎么样呢？

现在我们已经准备好接受第二个经典的问题了："基类的析构函数应当为虚函数吗？"

正如前面已经提到的，通常的答案是："哈？基类的析构函数当然应该是虚的！"只可惜这个答案是错误的，而且 C++标准库本身当中就包含了这方面的例证，只不过这个答案平常总是正确到能够给人一种正确的假相的地步。

一个不那么常见，但更为贴切一点的答案是："哈？基类的析构函数应当在你需要通过指向基类的指针来多态地析构派生类对象的时候为虚的！"这个答案从技术上来说的确没错，但仍然不够令人满意。

最近我得出的结论是，完全正确的答案如下：

准则：基类的析构函数要么应当为公用虚函数，要么应当为保护的非虚函数。

下面就让我们来看看为什么说这种做法是对的。

首先，很明显任何通过基类接口来进行的、应当具有多态行为的操作都应当是由虚函数来提供的。即便对于非虚接口来说也是如此，因为尽管其公用函数是非虚的，但幕后真正的工作却是由非公用虚函数来完成的，这样我们才能够得到我们所需的多态行为。

因此，如果我们可以通过基类接口来多态地销毁派生类对象的话，那么这种行为就必须是多态的，承担该行为的函数当然也就应当为虚函数了。实际上，语言对这一点有强制要求：如果你试图通过一个非虚析构函数来进行多态析构的话，你就会遇到"未定义行为"这个噩梦。因此： 135

```
// 示例18-3: 很明显需要虚析构函数。
```

```
//
class Base { /*...*/ };
class Derived : public Base { /*...*/ };

Base* b = new Derived;
delete b;                          // Base::~Base 最好是虚函数!
```

注意，这里的析构函数只是非虚接口模式所无法应用到的一个特例。为什么这么说呢？因为一旦执行流到达了基类的析构函数体中，就意味着其任何派生类对象的部分都已经被销毁，不复存在。此时如果基类析构函数体再去调用一个虚函数，那么虚函数分发机制有可能就会无法将该调用向继承树的下游（即派生类）转发了。简而言之，当执行流位于某个析构函数体（或构造函数体）当中的时候，其任何派生类对象都已经被析构而不复存在了（或还未诞生，对于构造函数的情况）。

然而基类并不总是需要支持多态销毁。例如，标准库自身当中的例子[C++03]：std::unary_function 和 std::binary_function。那两个类模板看起来像这样：

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg    argument_type;
  typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1    first_argument_type;
  typedef Arg2    second_argument_type;
  typedef Result result_type;
};
```

这两者是特地为了被实例化为基类（从而将其中包含的那几个标准的 typedef 插入到派生类当中）而作的，而且这两个类模板并没有提供虚析构函数，因为它们并不被用于多态销毁。这就是说，像下面这样的代码并不仅仅是不被认可的，而是完全非法的，你应当假设这种代码永远也不会被编写出来：

```
// 示例18-4: 有问题的代码，你可以假定这种代码永远也不会被编写出来。
//
void f(std::unary_function* f) {
  delete f;                        // 错误，非法的!
}
```

136

注意，标准明令禁止示例 18-4 中的做法，说如果你通过一个指向 std::unary_function 的指针来析构它的派生类对象的话，你将会遭到未定义的行为，然而标准并没有要求编译器去阻止程序员这么做（真遗憾！）。其实只要为 std::unary_function（或其他类似的类）添加一个保护的空析构函数就可以达到很好的防御效果，这么一来等于要求编译器侦测出这类错误并把错误信息扔给

不守规矩的家伙，这样做并不会破坏任何现存的符合标准的 C++代码。或许我们会在标准的下一个修订版中看到相应的改动，或许不会，但无论如何让编译器拒绝这种错误用法（而不是在标准里面纸上谈兵）肯定是有好处的。（是的，让析构函数成为非公用即意味着你不能够直接创建 unary_function 的对象，但这并不要紧，因为 unary_function 本来就只有在作为基类的时候才是有用的。）

最后，如果一个基类是具体类（concrete class，即其本身可以有实例的类），同时它又希望支持多态析构的话，该怎么办呢？是不是说它的析构函数得是公用的，因为如果不这样的话我们就无法容易地创建它的对象了？这样做是可以的，只不过倘若你这么做了的话你就已经违背了另一条原则：*不要继承自具体类*。或者正如 Scott Meyers 在[Meyers96]的条款 29 所说的："让继承树中的非叶节点成为抽象类。"[诚然，这在实践当中是可能发生的（当然，是在别人而不是你编写的代码中），而在我们所讨论的这个特定的例子当中，你或许得采用一个公用的虚析构函数来弥补一个已经比较糟糕的设计。不过，如果可以的话，你最好还是进行一点重构以改正你的设计。]

那么，简而言之剩下的情况就是二选一：（a）你想要允许通过基类指针来进行多态析构，这种情况下基类的析构函数必须为公用虚函数；（b）否则虚函数就应当为保护且非虚的，保护能够防止意外的误用。

小结

总的来说，你应当尽量将基类的虚函数放在私有区段（如果的确有必要也可将其放在保护区段）。这样做能够将接口与实现分离开来，有助于接口的稳定，并易于日后对幕后实现机制进行改动或重构。对于常规的基类函数来说：

- 准则之一：尽量利用非虚接口模式让接口函数成为非虚的。
- 准则之二：尽量让虚函数成为私有的。
- 准则之三：只有当派生类需要对基类中某个虚函数的实现进行调用的时候才将对应的虚函数设为保护的。

以下原则仅对析构函数这个特例有效： 137

- 准则之四：基类的析构函数应当要么为公用虚函数，要么为保护非虚函数。

确实，标准库本身也并没有完全遵循这些设计原则。这也正从一个侧面反映了 C++界这些年来确实学到了新的法则。 138

第 19 条　对派生类施加规则

难度：5

身居继承树的顶端并不意味着你可以阻止使用派生类的程序员犯低级错误，这种情况再平常不过了。不过有时候你倒的确能够做到！本条讨论基类可以在设计时作一些安全方面的考虑从而让使用派生类的程序员更不容易犯错。

JG问题

1. 编译器什么时候会为一个类隐式声明和隐式定义如下这些函数，隐式声明/定义出来的版本具有什么样的语义？请逐一给出明确答复，并指出在什么情况下隐式定义出来的版本是不合法的（ill-formed）。

（a）默认构造函数

（b）复制构造函数

（c）复制赋值操作符

（d）析构函数

2. 对如下的类 X 来说具体有哪几个函数是被隐式声明和隐式定义的？它们的函数签名分别是什么？

```
class X {
 auto_ptr<int> i_;
};
```

Guru问题

3. 假设你有一个基类，你要求它的所有派生类都不使用它的隐式声明/定义出来的函数。例如：

```
class Count {
public:
 // Count的作者在这里注明了所有派生类都应当采用虚继承，并且
 // 它们的所有构造函数都只该调用Count类中特意准备的这个构造函数。
 //
 Count(/* 一些特殊的参数 */);
 Count& operator=(const Count&);                    // 常规
 virtual ~Count();                                  // 常规
};
```

139

然而遗憾的是，程序员也是人，他们有时候也会忘记应该显式地定义其中两个函数。

```
class BadDerived : private virtual Count {
 int i_;

 // 默认构造函数：应当调用Count类的特殊的构造函数，但它做到了吗？
```

```
  // 复制构造函数: 应当调用Count类的特殊的构造函数，但它做到了吗?
  // 复制赋值:没问题?
  // 析构函数:没问题?
};
```

在我们的这个例子的上下文当中，Count 的作者有没有办法迫使派生类进行正确的编码呢？换句话说，如果派生类没有正确编码的话，会引发一个编译期（最好）或运行时（至少）的错误吗？

更一般的说法是：基类作者有办法迫使派生类作者显式地编写出这 4 个基本操作吗？如果可以的话，怎么做？如果不可以的话，为什么？

解决方案

隐式生成的函数（或者说编译器为你代劳的工作）

在 C++中，有 4 个类成员函数可以被编译器隐式生成，分别为：默认构造函数、复制构造函数、复制赋值操作符以及析构函数。

这么规定的原因是为了方便组合以及向后兼容于 C。C 风格的**结构**是只包含公用数据成员的类；特别地，它们不包含任何（显式定义的）成员函数，当然你得能够创建、复制以及销毁它们。为了使之成为可能，C++语言规定，如果你自己没有定义某些必要的操作的话，编译器会自动生成一些适当的函数（或这些函数的某个子集）以完成适当的工作。

本条正是要讨论所谓的“适当”具体是指什么：

1. 编译器什么时候会为一个类隐式声明和隐式定义如下这些函数，隐式声明/定义出来的版本具有什么样的语义？请逐一给出明确答复，并指出在什么情况下隐式定义出来的版本是不合法的。 140

简而言之，一个隐式声明的函数仅当你实际试图去调用它的时候才会被定义出来。例如，一个隐式声明的默认构造函数，仅当你试图在无构造函数参数情况下构造其所属类的对象时才会隐式定义。

那么，为什么我们要区分隐式声明跟隐式定义的时机呢？这样规定有何用处？实际上，由于这些函数可能不会被调用到，所以如果它们永远也不被调用到，那么即便其隐式定义生成出来可能是非法的，但由于实际上并未生成，因而也就不会影响程序的合法性。

为了简便起见，在本条的讨论当中，除非特别注明，凡是提到“成员”，我都是指“类的非静态数据成员”。同时我也会提到“隐式生成”一词，它是“隐式声明并定义”的简称。

隐式声明的函数的异常规格（Exception Specification）

对于可能被隐式声明的所有 4 种特殊函数，编译器会将它的异常规格尽量放宽，以便允许其隐式定义所调用到的函数可能会抛出的所有异常。例如：

```
// 示例19-1(a)
//
class C // ……
{
 // ……
};
```

由于 C 当中没有显式声明任何构造函数，因此编译器会隐式生成一个默认的构造函数，后者的语义为调用所有基类以及成员子对象的默认构造函数。因而其隐式生成的默认构造函数的异常规格必须允许所有基类及成员默认构造函数可能抛出的所有异常。如果其中有任何基类或成员的默认构造函数不带有异常规格的话，就意味着隐式生成的构造函数可能会抛出任何异常：

```
// 公用的:
inline C::C();          // 可能抛出任何异常
```

如果 C 的每个基类或成员都具有一个带有显示异常规格声明的默认构造函数，那么 C 的隐式生成的默认构造函数将可能抛出前者的异常规格列表里面列出的任何异常：

```
// 公用的:
inline C::C() throw (
 // C的基类或成员的默认构造函数可能抛出的任何异常;
 //即C的基类或成员的默认构造函数的异常规格列表的并集
                    );
```

141

似乎这里有一个潜藏的陷阱。考虑这么一种情况：如果隐式生成的函数中有一个重写了基类对应的虚函数的话会出现什么情况？这对构造函数来说不会发生（因为构造函数永远也不是虚函数），然而对于复制赋值操作符来说则是有可能会发生的（如果你故意在基类当中设置一个签名跟派生类中自动生成的复制赋值操作符签名一模一样的虚函数的话），而对于析构函数则更是如此：

```
// 示例19-1(b): 危险!
//
class Derived;

class Base {
public:
 // 这个例子有点故意捏造的意思，如果果真出现这种情况几乎肯定是
 // 糟糕的做法，不过从技术上来说，使用这种技巧来声明一个与派生类中的
 // 复制赋值操作符的签名一模一样的基类赋值操作符是可行的;
 // 不过在动下面这样的念头之前记得一定要先阅读[Meyers96]中的第33条:
 //
 virtual Base& /* 或Derived& */
 operator=(const Derived&) throw(B1);

 virtual ~Base() throw(B2);
};
```

```
class Member {
public:
  Member& operator=(const Member&) throw(M1);
  ~Member() throw(M2);
};

class Derived : public Base {
  Member m_;

  // 隐式声明了4个函数:
  // Derived::Derived();                                          // 没问题
  // Derived::Derived(const Derived&);                            // 没问题
  // Derived& Derived::operator=(const Derived&) throw(B1, M1);   // 错
  // Derived::~Derived() throw(B2, M2);                           // 错
};
```

那么问题出在哪呢？问题是上面代码中标明的两个函数是非良构的，因为不管什么时候只要你重写了继承的虚函数，那么你的派生类函数的异常规格就必须至少跟对应的继承的虚函数一样严格。这样规定是有意义的：因为如果不这样的话就意味着那些通过基类指针来调用派生类虚函数的代码可能会得到一个不在基类相应虚函数的异常规格列表中的异常。例如，假设示例 19-1(b) 中的情况是允许的，那么考虑如下的代码：

142

```
Base* p = new Derived;

// 哎呀——这里可能会抛出B2或M2，而不像Base::~Base
// 承诺的那样最多只会抛出B2:
delete p;
```

这是关于“为什么每个析构函数应当要么具有空异常规格（throw()，不抛异常），要么根本没有异常规格（可能抛出任何异常）”这一原则的另一个正面论据。此外，析构函数应当永远不抛出任何异常，而且你在编写析构函数的时候应该当它具有一个空的异常规格列表似的（即便其函数声明上并没有 throw()的字样）。（参见 *Exceptional C++* [Sutter00]的第 12 条，其中有一节为“抛出异常的析构函数以及为什么说它们是危险的”）。

准则：永远不要让异常从析构函数中跑出来。在编写任何析构函数的时候总是就当它具有一个不抛任何异常的异常规格声明那样。
永远不要为函数编写异常规格声明。或许你可以编写一个空的，不过如果我是你的话，我连这个也会避免（见第13条）。

这也是关于当心虚赋值操作符的另一个证据。参见[Meyers96]第 29 条及[Dewhurst03]第 76 条中关于虚赋值操作符的更多危害，以及如何避免它们。

准则：避免将赋值操作符设为虚函数。

现在，让我们逐一来考虑上文提到的 4 个能被隐式生成的函数：

隐式默认构造函数

（a）默认构造函数

默认构造函数仅在你自己不声明任何构造函数的时候才会被隐式声明。一个隐式声明的默认构造函数是**公用内联的**。

一个隐式声明的默认构造函数只有在你实际试图调用它的时候才会被隐式定义，其隐式定义的效果就好像是你自己写了一个空的默认构造函数那样，可能抛出其基类或成员的默认构造函数可能抛出的任何异常。如果你自己手写的那个空的默认构造函数是非法的，那么该隐式定义就是非法的（例如，如果某个基类或成员没有默认构造函数的话它就是非法的）。

143

隐式复制构造函数

（b）复制构造函数

如果你不自己声明一个复制构造函数的话，编译器就会为你自动隐式声明一个。隐式声明的复制构造函数是**公用内联的**，编译器会尽可能地将其形参声明为 const 引用（其形参为 const 引用当且仅当其每个基类和成员都具有按 const 或 const volatile 引用来接受参数的复制构造函数），实在不行才将形参声明为 non-const 引用。

这就对了，就像大多数 C++程序员那样，许多时候标准自身也基本都会忽略 volatile 这个关键字。尽管编译器会自发地为隐式声明的复制构造函数（以及复制赋值操作符）的形参加上 const 修饰（如果可能的话），不过坦白地说对于是否会添加 volatile 修饰倒是语焉不详。哦，好吧，不过这就是生活。

一个隐式声明的复制构造函数会在你实际试图调用它去复制给定类型的对象时被隐式定义出来，对其所有基类和成员子对象进行逐成员复制，并可能抛出它的基类或成员复制构造函数可能抛出的所有异常。如果有任何基类或成员具有不可访问或二义性的复制构造函数的话，该定义就是非法的。

隐式复制赋值操作符

（c）复制赋值操作符

如果你自己不声明任何复制赋值操作符的话，编译器会为你隐式生成一个。隐式声明的复制赋值操作符是**公用内联的**，会返回被赋值对象的 non-const 引用，并尽可能会按 const 引用接受参数（当且仅当其所有基类及成员的复制赋值操作符都是按 const 引用传参时该隐式生成的复制赋值操作符的形参才是 const 引用的，否则就是 non-const 引用）。同样，标准在这里对 volatile 也语焉不详。

一个隐式声明的复制赋值操作符仅当你实际需要调用它（对给定类型的对象进行赋值）时才

会由编译器隐式定义出来，其语义是对所有基类及成员子对象进行逐成员赋值（包括对虚基类子对象可能存在的多次赋值），并且可能抛出其基类或成员的复制赋值操作符可能抛出的所有异常。如果其任何基类或成员为 const、为引用或者具有不可访问或二义性的复制赋值操作符的话，该隐式生成的复制赋值操作符就是非法的。 144

隐式析构函数

（d）析构函数

如果你自己不去声明任何析构函数的话，编译器就会自动为你隐式地生成一个。一个隐式声明的析构函数是**公用内联的**。

一个隐式声明的析构函数仅当你实际试图调用它的时候才会被隐式定义出来，其效果等同于你自己写了一个空的析构函数一样，它可能抛出其基类或成员的析构函数可能抛出的任何异常。如果其任何基类或成员具有不可访问的析构函数，或者如果有任何基类析构函数为**虚**且并非所有基类及成员析构函数都具有同样的异常规格的话，该隐式生成的析构函数就是非法的（参见上文关于“隐式声明的函数的异常规格”的描述）。

一个auto_ptr成员

2. 对如下的类 X 来说具体有哪几个函数是被隐式声明和隐式定义的？它们的函数签名分别是什么？

```
// 示例19-2
//
class X {
 auto_ptr<int> i_;
};
```

题外话：这个例子很好地说明了语言在隐式声明/定义 4 个特殊成员函数方面的规则。这并不是个关于良好编码风格的例子。总的来说你应当避免将 auto_ptr 作为成员并尽量避免使用 auto_ptr。你应当尽量使用 shared_ptr，shared_ptr 目前已经被加入了 C++标准库。

在类 X 中，如下函数被隐式声明为公用成员。当你试图调用它们的时候，编译器就会将它们隐式定义出来，效果如下：

```
inline X::X() throw()                          : i_() { }
inline X::X(X& other) throw()                  : i_(other.i_) { }
inline X& X::operator=(X& other ) throw()      { i_ = other.i_; return *this; }
inline X::~X() throw()                         { }
```

这里，隐式生成的复制构造函数及复制赋值操作符之所以接受 non-const 引用的参数是因为 auto_ptr 中的相应函数也是这么做的。类似的，所有这些函数都具有不抛任何异常的异常规格声明，因为 auto_ptr 中相关的函数也没有哪个会抛出异常，而且实际上 auto_ptr 的任何操作都不能抛出异常。 145

注意，上面的复制构造函数以及复制赋值操作符会导致 auto_ptr 成员的指针所有权的传递。后者可能是 X 的作者不愿意看到的，因此对于 X 来说自己编写这些函数通常是有必要的。（参见 *More Exceptional C++* [Sutter02]的条款 30 跟 31 中更多关于这方面相关主题的详细讨论。）

“逆子”以及其他“家庭”问题

3. 假设你有一个基类，它要求其所有派生类都不使用隐式声明和定义的特殊函数。例如：

```
// 示例19-3
//
class Count {
public:
// Count的作者在这里注明了所有派生类都应当采用虚继承，并且
// 它们的所有构造函数都只该调用Count类特意准备的这个构造函数。
//
Count(/*一些特殊的参数 */);
Count& operator=(const Count&);        // 常规
virtual ~Count();                      // 常规
```

这里我们有这样一个类：它想要它的任何派生类都乖乖听话地调用它特地准备的构造函数，或许这样一来 Count 才可以对系统中创建的派生类对象的数目进行跟踪。这是一个要求进行虚继承的有说服力的理由，因为虚继承可以避免在一些特殊情况下（譬如某个派生类中具有多个 Count 类的子对象[1]）发生重复计数的情况。

有趣的是，你有没有注意到 Count 中可能已经存在了一个设计错误？它有一个隐式生成的复制构造函数，这对于跟踪计数的正确性也许没什么好处。我们只需将它声明在私有区段且不加定义就可以将它禁用掉：

```
private:
 // 未定义，无复制构造函数
 Count(const Count&);
};
```

146

那么，这就是说 Count 想要它的派生类们乖乖听话。然而这些家伙并不总是会循规蹈矩，对不对？实际上我们无需搜肠刮肚就可以轻易找出一个不听话的家伙来：

遗憾的是，程序员也是人，他们有时也会忘记他们应当将这两个函数显式编写出来：

```
class BadDerived : private virtual Count {
 int i_;
 // 默认构造函数：应当调用Count特地准备的构造函数，但事如人愿吗？
```

1. 本例是根据 Marco Dalla Gasperina 在他的未发表的“对象计算和虚继承”一文中的内容改写的。他的代码中没有我下面要讨论的设计错误。实际上他的那篇文章并不是关于对派生类施加规则的，而是关于一些其他内容的，但是他在其中给出的例子让我不由地想：“嗯，我怎么才能防止派生类作者在使用我的 Count 计数基类的时候忘记调用我特地准备的构造函数呢？”

简单地说，未如人愿，BadDerived 的默认构造函数非但不会调用 Count 特地准备的构造函数，甚至还面临一个更为根本的问题：究竟存不存在这么一个 BadDerived 的默认构造函数呢？答案或许会让你心里感到不塌实，答案是：某种程度上。BadDerived 的确具有一个隐式声明的默认构造函数（没问题），然而一旦你试图去调用它的话，程序就会成为不合法的（糟糕！）。

让我们来看一看为什么会这样。首先，BadDerived 并不会定义任何它自己的构造函数，因此一个默认的构造函数就会被隐式声明出来。这蛮好，没问题。但是当你试图使用该构造函数的时候[即当你试图创建一个 BadDerived 对象（你可能认为创建其对象是相当重要的能力，你的想法也的确没错）的时候]，编译器就会为这个隐式声明的构造函数生成定义，或者说至少编译器应当在此时这么做，然而由于这个隐式定义应当调用基类的默认构造函数，后者又根本不存在，所以程序就成不合法的了。至少，任何试图创建 BadDerived 对象的程序都不是符合 C++标准的程序，因此我们应当将责任归咎于 BadDerived。

那么，关于“BadDerived 究竟存不存在一个默认构造函数呢？”这个问题，答案是“某种程度上”。实际上编译器声明了它，但你不可以去调用它，这令它变得形同虚设。想想看，如果一个家庭里的孩子都像这样的话，这个家庭能算是愉快的家庭吗？。

```
// 复制构造函数: 应当调用Count特地准备的构造函数，但它做到了吗?
```

类似的，隐式生成的复制构造函数也会被声明，但在定义时并不会调用 Count 中特定的构造函数。如果这里的 Count 类就是原来那个 Count 类的话，BadDerived 的复制构造函数就会调用 Count 的隐式生成的复制构造函数。

如果我们决定抑制 Count 的隐式生成的复制构造函数（参考上文），那么 BadDerived 虽然具有一个隐式声明的复制构造函数，但由于它不能被隐式定义（因为编译器会发现 Count 的默认复制构造函数是不可访问的），所以任何使用它的企图都会使程序变成非法的 C++代码。 147

值得庆幸的是，现在情况终于开始出现转机了：

```
// 复制赋值: 没问题吗?
// 析构函数: 没问题吗?
};
```

是的，隐式生成的复制赋值操作符及析构函数的行为都是正确的，即调用（对于析构函数来说则是重写）基类中相应的函数。因此好消息是至少出现正面的情况了。

不过这个类家族当中的成员仍然不开心。毕竟，每个家庭都会有点家规的。我们就不能给孩子们一点玩具让他们安静下来吗？

对派生类施加规则

在我们的这个例子的上下文当中，Count 的作者有没有办法迫使派生类进行正确的编码呢？换句话说，如果派生类没有正确编码的话，会引发一个编译期（最好）或运行时（至少）的错误吗？

办法并不是去禁止编译器为派生类隐式声明特殊函数（我们还做不到这一点），而是让编译器试图进行的隐式定义成为非法的，从而导致编译器给出一个可以理解的错误。

更一般的说法是：基类作者有办法迫使派生类作者显式地编写出这4个基本操作吗？如果可以的话，怎么做？如果不可以的话，为什么？

好了，随着我们逐步认识到在这四个基本操作的隐式声明和定义的过程当中发生了什么，我们逐渐开始注意到“不可访问”及“二义性”这两个字眼。一种做法是通过为基类的某些特殊函数添加具有二义性的重载版本（即便具有不同的访问限定）[1]从而阻止编译器为其派生类隐式定义对应的特殊函数，不过这种做法似乎起不到多大帮助。最简单的做法是将基类中相应的函数声明（定不定义随你）在私有区段从而有选择性地禁用它们，而且这种做法对虚函数之外的所有其他函数皆有效。

```
// 示例19-4: 迫使派生类不能使用它们隐式生成的函数，
// 做法是让Base中相应的函数变成不可访问的。
//
class Base {
public:
  virtual ~Base();

private:
  Base(const Base&);                    // 无定义
  Base& operator=(const Base&);         // 无定义
};
```

148

这里的 Base 类没有默认的构造函数（因为一个用户自定义的构造函数已经被声明了，从而抑制了编译器的自动生成），同时 Base 具有一个隐藏的复制构造函数和复制赋值操作符。但我

1. 例如：

```
class B
{
public:
B(){}
B(B const&){}
B(B const&,int=0){}
};

class D:public B
{
};

int main()
{
D d1; // 没问题
D d2 = d1; // 错，编译器试图定义D的隐式声明的复制构造函数，却发现B的复制构造函数调用具有二义性！
}          //
```

——译者注

们无法将析构函数也藏起来，因为它必须对派生类总是可见的，而且它通常应当为公用虚函数（如示例 19-4 所示），或者为保护非虚的（见第 18 条）。

我们的想法是，即便我们确实想要支持某个给定操作（例如复制赋值）的某种形式，如果我们无法通过常规函数（operator =）来完成它，那就让常规函数“靠边站”（放至不可访问区段），并提供一个具名的，或者其他容易区分的函数来照我们的意愿来完成工作。

这种想法最终将我们带到了哪里呢？让我们来看看：

```
class Derived : private Base {
int i_;

 // 默认构造函数:      声明了，但定义是不合法的
 //                   (因为不存在Base的默认构造函数)

 // 复制构造函数:      声明了，但定义是不合法的
 //                   (因为Base的复制构造函数是不可访问的)

 // 复制赋值操作符:    声明了，但定义是不合法的
 //                   (因为Base的复制赋值操作符是不可访问的)

 // 析构函数:          合法，将会通过编译
};
```

唔，还不赖……最多可能有 4 个编译期错误而我们得到了 3 个，而且似乎这已经是我们所能做到的极限了，或者说我们其实只需做到这一步即可。

这个简单的解决方案无法解决析构函数的问题，但这其实无关紧要，因为相较之下我们不大可能使用一个特殊意图的成员函数来替换析构函数；基类的析构函数必须永远得到调用，这可不像构造函数可以有多个重载版本，析构函数没有多条路可走（永远只有惟一一个析构函数）。困难的部分是如何确保基类的那些特地准备的构造函数能够被正确地调用起来从而正确地初始化基类子对象；这样基类子对象才有足够的信息以备日后能正确地析构。

这还不赖。简单的解决方案通常就意味着最好。本例其实还有一些更为复杂的解决方案；下面就让我们来简略地巡视一下它们，以便打消我们心中的疑虑（确信它们在析构函数或其他任何方面也没什么优势）。

*替代方案之一：让基类中相应的函数成为二义性的。*这种做法没任何优势，它仍旧没有禁止隐式生成的析构函数，而且工作量也更大。

149

*替代方案之二：让基类中相应的函数在运行时出错。*例如我们可以让它们抛出 std::logic_error 异常。但这种做法也没有阻止编译器隐式生成析构函数（在不破坏任何可行的析构函数的前提下），而且它还将编译期错误推迟到了运行时，这可不太妙。

准则：尽量用编译期错误来取代运行时错误。

替代方案之三：将基类中相应的函数设为纯虚函数。这毫无用处：它对构造函数一点作用也不起（不管是默认构造函数还是复制构造函数）；对复制赋值操作符也没什么帮助，因为派生类的复制赋值操作符跟基类中的具有不同的签名（参数类型不同）；而且它对析构函数也不起作用，因为仍然满足编译器为派生类生成默认析构函数的条件。

替代方案之四：使用不具备默认构造函数的虚基类。这迫使最上层派生类构造函数负责调用该虚基类的构造函数（从而出错）[1]。该方案在两种构造函数身上有运用前景，另外还有一个优势就是对并非直接继承自 Base 的类也能起作用，因此这是惟一一个能和示例 19-4 中的解决方案结合使用的方案。不过这种做法只对构造函数有效，派生类的隐式生成的复制赋值操作符和析构函数仍然未被禁止。

小结

为了阻止编译器为派生类隐式生成默认构造函数、复制构造函数或者复制赋值操作符，最简单的、最佳的选择是将基类中相应的函数“藏”在非公用区段（或者干脆让它消失，例如默认构造函数）。

150

1. C++标准要求最顶层派生类负责调用虚基类的构造函数。这一规则是为了避免只有一份的虚基类子对象被重复构造多次。——译者注

内存和资源管理

如果说还有什么话题 C/C++程序员最关心，那就是内存和资源管理了。与其他语言相比，C++最强大的一面就是它赋予了程序员控制和管理内存及其他资源的能力，尤其是能够通过使用标准库容器来有选择性地自动化内存管理。

你对不同的标准容器的实际内存开销有多少认识？你能肯定一个内含 1 000 个对象的 list 会比同等的 set 消耗更少的内存空间吗？接下来说到异常：使用 nothrow new 是不是有助于令你的代码更异常安全呢？最后，在现实中的许多流行的平台上,什么时候可以不用担心 new 是否失败(喘口气!)，为什么呢？

第 20 条 内存中的容器之一：内存管理的层次

难度：3

现代操作系统上的内存管理就其本身来说是复杂而成熟的，然而对于C++程序它也只不过是一个内存管理层次。标准库则提供了更多的层次让你选择，这些层次对C++程序更加重要。

JG问题

1. 内存管理器（也称内存分配器）是指什么？其基本功能是什么？简要地描述 C++中的两个主要的动态内存管理策略。

Guru问题

2. 在 C++标准库以及使用了 C++标准库实现的典型环境中存在有几种不同层次的内存管理？它们相互之间的关系有什么特点？它们之间如何交互？它们是如何各司其职的？

解决方案

本条及下一条旨在解决第 21 条中的问题 2：给定同一类型同样数目的一组对象，各种标准库容器各消耗多少内存去存放它们？

然而，为了解答这个问题，我们首先得踏过动态内存管理沼泽的边缘，然后再回顾一点数据结构的知识。具体而言，我们得考虑如下两个主要问题：

- vector、deque、list、set/multiset 以及 map/multimap 这类容器内部使用的数据结构；
- 动态内存分配的工作方式。

152 让我们先来回顾一下动态内存分配，然后再回过头去看看它们在标准库中是如何运用的。

内存管理器及其策略：概观

1. 内存管理器（也称为内存分配器）是指什么？其基本功能是什么？简要地描述 C++中的两个主要的动态内存管理策略。

为了弄清各种容器的内存开销，很重要的一个前提是理解底层的动态内存分配是如何进行的，毕竟容器总得从某个内存管理器那里获取内存的，而内存管理器又必须根据某种内存管理策略来确定如何划分出可用的内存。

简而言之，C++中存在着如下两个流行的内存管理策略。更进一步的细节超出了本条款讨论的范畴；你可以找一本最喜爱的操作系统教科书来获取更多的信息：

- 其一是通用意图（general-purpose）的分配策略，它可以提供调用方可能要求的任意大小（称为请求大小或块大小）的内存块。通用意图的分配策略非常灵活，但有好几个缺点，其

中两个就是：（a）性能，因为这种分配策略需要做更多的工作；（b）碎片，因为随着内存块的不断分配和释放，最终可能会导致许多小的、不连续的未分配内存区域。

- 其二是固定大小（fixed-size）的分配策略，它总是返回固定大小的内存块。这显然不像通用意图的分配策略那样灵活，但它的好处是可以在短得多的时间内完成分配，而且不会产生那样的内存碎片。

第三种主要的分配策略就是垃圾收集，该策略与 C/C++指针、malloc 和 new 并不完全兼容，因此与我们当前讨论的主题不直接相关。然而，垃圾收集技术正在变得越来越流行，而且正在走进 C++，只不过不是跟指针和 new 搭配；我打算在以后的书中详细描述这方面的内容。（想要知道更多关于垃圾收集本身的问题以及 C++与垃圾收集相结合的一些讨论，可参见[C++CLI04]及[Jone96]。）

实际上，你通常会见到这些策略是被组合在一起运用的。例如，或许你的内存管理器对任何大于 S 的内存请求使用的都是通用意图的分配策略，而对小于等于 S 的请求则一律采用固定大小的分配策略以获得优化的效果。一般来说，为大小为 1、为 2、为 3，等等的内存请求全都特殊对待的做法是不实用的，所以内存管理器一般会基于某个特定大小（如 16 字节）的倍数维护一个个的所谓仓库（arena）[1]。譬如说，你请求 16 字节的内存，那么好极了，因为你恰好只使用 16 字节；但如果你请求 17 字节的内存，这块内存就会从“存放”着 32 字节大小的内存块的 arena 中分配出来，也就是说浪费了 15 字节。这是内存管理策略可能导致的额外开销的来源之一，我们待会儿再来讨论这个。

很明显下一个问题是：谁负责选择内存管理策略？ 153

划分策略

2. 在 C++标准库以及使用了 C++标准库实现的典型环境中存在有几种不同层次的内存管理？它们相互之间的关系有什么特点？它们之间如何交互？它们是如何各司其职的？

答案是存在着好几种可能的内存管理层面，其中每层都可能会覆盖原先的（较低的）层面：

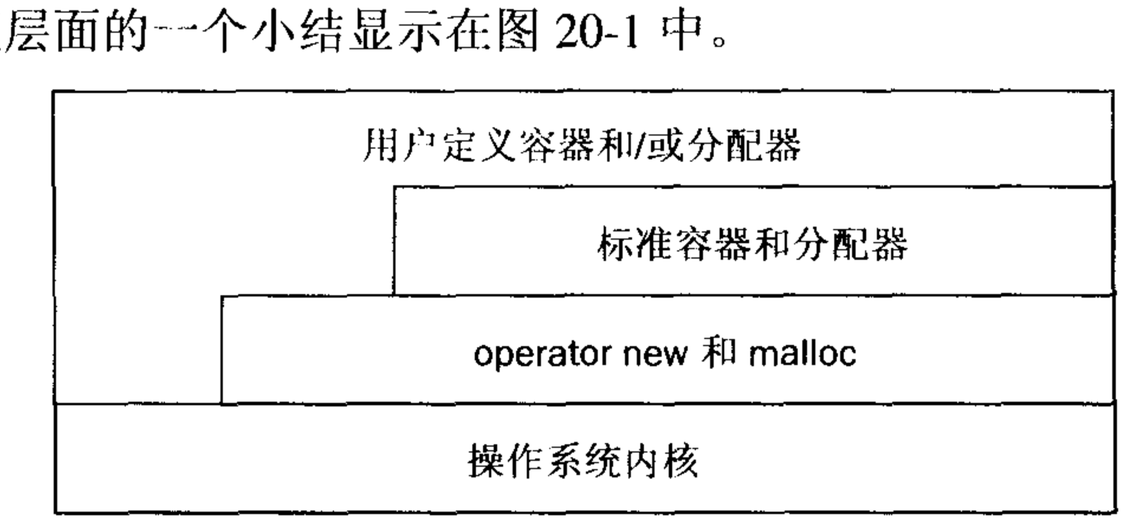
- 首先是操作系统内核提供了最为基础的内存分配服务。这个底层的分配策略及其特性可能根据操作系统平台的不同而有很大差异，而且这一层内存管理是最可能受到硬件相关的考虑所影响的。
- 编译器的默认运行时库也会建立起它自己的内存分配服务，譬如 C++中的 operator new 和 C 中的 malloc，这一层是建立在操作系统的本地分配服务基础上的，可能只是后者的一层薄薄的外衣，继承了其特性；也有可能通过先从后者那里“买进”大块的内存（chunk）

1. 一般的实现是采用一个数组（链表也可），其每个元素都是一个指针，指向一个可用内存块的链表（称为 arena）。如，数组的第一个元素代表 16 字节的 arena，所以它所指向的空闲内存链表内的内存块大小全是 16 字节。第二个元素代表 32 字节（16*2）的 arena，因而它所指向的空闲内存链表内的内存块的大小皆为 32 字节。依此类推……——译者注

然后再切割转手分配出去的方式来覆盖操作系统的本地内存分配服务，至于具体如何切割“买来的”大块内存就是由这一层运行时库自己自由选择的了。

- 而标准库容器和标准分配器进而又利用（也许会覆盖）运行时库提供的服务来实现它们自己的策略和优化。
- 最后，用户自定义的容器或用户自定义的分配器可能会进一步复用上面提到的任何一层服务（例如，如果可移植性并不那么重要的话，它们可能会想要直接访问本地分配服务），并按照它们自己的意愿来进行定制。

关于这些内存管理层面的一个小结显示在图 20-1 中。



154

图 20-1 内存管理的主要层面。典型情况下每层都可以由比它更低的相邻层面来实现

因此，内存分配器有好些不同的形式和种类，而且在不同的操作系统上，或者相同操作系统上的不同编译器间，或者不同容器间都会有所不同，甚至即便是在不同的对象间也会有所不同，譬如说 vector<int>使用 allocator<int>而 vector<int, MyAllocator>的 MyAllocator 则可以是任何你喜欢的分配策略。

准则：弄清谁负责做什么：弄清你当前的平台和标准库所使用的实际（且系统相关的）分配策略以及它们各自的职责是什么。

小结

现代操作系统上的内存管理就其本身来说可能是复杂而成熟的，然而充其量它也只不过是（仅仅）一层（值得 C++程序员关心的）内存管理层而已。标准库则提供了更多的层次让你选择：首先是它的用于分配和归还的基本设施，然后是它的标准容器和分配器，再然后就是你自己编写的名目繁多五花八门的自定义容器了。

然而当你请求内存的时候，你是否知道你究竟得到了什么呢？其开销又是多少呢？标准库容器究竟耗用了多少内存呢（理论上的以及实际上的）？请听下条分解。

155

第 21 条　内存中的容器之二：它到底有多大　难度：3

当请求内存时你实际得到的是什么，对此你究竟有多少认识？其实际开销是多少？标准容器使用了多少内存（理论上、实际上、以及你今天下午将要编写的程序中）？

JG问题

1. 当你使用 new 或 malloc 请求 *n* 字节的内存时，你实际只消耗了 *n* 字节内存吗？为什么？为什么不？请解释。

Guru问题

2. 给定同一类型同样数目的一组对象，各种标准库容器各消耗多少内存去存放它们？

解决方案

“我要用operator new请求200字节”

1. 当你使用 new 或 malloc 请求 *n* 字节的内存时，你实际只消耗了 *n* 字节内存吗？为什么？为什么不？请解释。

当你使用 new 或 malloc 请求 *n* 字节内存时，你实际上至少耗用了 *n* 字节的内存，因为一般情况下内存管理器必须加上一些额外的开销。一般来说包括两项额外开销：系统维护（housekeeping）开销以及大块内存导致的开销。

先来考虑第一项开销：在一个通用意图（即不是固定大小的）的分配策略中，内存管理器必须以某种方式记住每块分配出去的内存块的大小，这样以后当这些内存块被归还（当你调用 delete 或 free 时）的时候才知道释放多大的内存。通常内存管理器会将实际分配的内存块的大小存放在该内存块的头部，而返回给你的指针就是从实际分配的内存块头部向后偏移系统维护信息所耗大小之后的指针（见图 21-1）。当然，这意味着它得耗用额外的空间来存放内存块的大小值，该值的最大值是可能分配出的最大空间大小，因此一般情况下它是一个指针大小[1]。在释放内存块的时候，内存管理器会首先通过你返还给它的指针寻找到低层细节信息，将指针向后偏移系统维护信息所占大小进而得到内存块的大小，然后才进行实际的内存释放。 156

1. 因为有效的内存分配大小不可能突破指针寻址能力的上限，例如典型的 32 位机器上指针的寻址极限是 2 的 32 次方，超过这个大小的内存就无法寻址了，因此低层细节数据的最大值也就是 2 的 32 次方，即耗用一个指针所耗用的相同大小的空间（32 位机器上是 4 字节）。——译者注

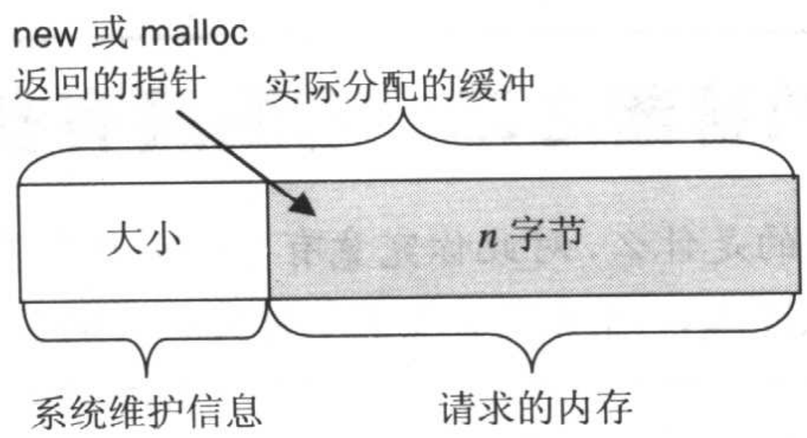


图 21-1　典型的内存分配使用通用意图的分配器来满足 *n* 字节的分配请求

当然，固定大小的分配策略（即总是返回给定大小的内存块）并不需要存放此类信息，从而节省了这笔开销，因为它们总是知道分配出去的内存块是多大的。

下面让我们来看一看所谓的大块内存开销：即便你并不需要存放额外的信息，通常某些内存管理器也会预留（reserve）出比你请求的空间更大的空间，因为内存通常是按照一定大小的 chunk 来分配的。

一方面，有些平台要求某些特定类型的数据满足特定的字节边界对齐条件（例如有些平台就要求指针只能存放在 4 字节边界上[1]），如果特定的条件不能满足的话就会影响访问内存的速度，或者干脆出错。这就是我们所谓的“对齐”（alignment），正是该现象导致了对象数据中存在着（可能是在尾部）填充字节（padding）。即便是旧式的内建 C 风格数组也会受到对齐需要的影响，因为对齐会影响到结构的大小（sizeof）。见图 21-2，其中我将尾部填充跟内部填充区分开了，尽管两者都会对结构的大小产生影响。

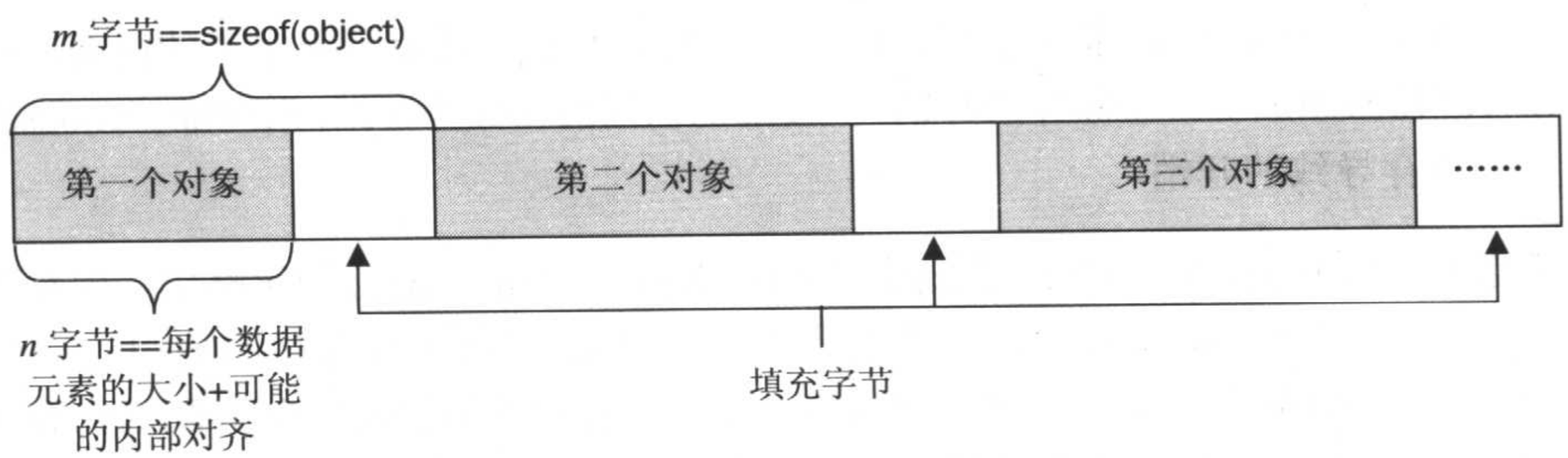


图 21-2　弄清所谓的“连续”的真相，图中是实际占用 *m* 字节的（注意：对象的大小为 *m* 字节）*n* 字节对象（即对象所有数据成员大小的和为 *n*）的数组在内存中的典型布局

157

例如：

```
// 示例21-1: 假设sizeof(long) == 4 并且long必须是4字节边界对齐的
//
struct X1 {
   char c1;      // 在相对于对象起始地址偏移量为0处，占用1个字节
                 // 后续三个字节皆为填充字节
```

1. 换句话说，指针变量的起始地址一定得是 4 的倍数。——译者注

```
    long l;          // 第5至第8个字节：占用4个字节，在4字节边界对齐
    char c2;         // 第9个字节:占用1个字节
                     // 后续三个字节皆为填充字节（见下文描述）
};// X1的大小为12
```

在图 21-2 中，n == 1+3+4+1 == 9，m == sizeof(X1) == 12[1]。注意，所有的填充字节都会计入 X1 的大小（sizeof(X1)）。而在尾部的填充字节可能看起来很奇怪，但实际上是需要的，因为这样才能保证当你创建 X1 的数组（它们在内存中是一个一个紧挨着存放的）的时候 X1 中的成员 l 总是在 4 字节边界对齐的。X1 尾部的填充字节是最值得注意的地方，也是最令第一次学习对象数据布局的程序员惊讶之处。当我们将 X1 的成员重新调整一下次序之后就会发现十分令人惊讶的结果：

```
// 示例21-2: 将X1的成员次序重新调整后的结果
//
struct X2 {
   long l;          // 前4个字节
   char c1;         // 第5个字节
   char c2;         // 第6个字节
                    // 后续两个填充字节
};// sizeof(X2) 为8
```

现在，该结构的所有成员就真的位于连续内存中了（n==6）[2]，不过尾部仍然有额外的填充，使得 m==sizeof(X2)==8，当你创建 X2 对象的数组时，尾部的填充是最值得注意的。最后两个字节就是图 21-2 中标明的填充部分。

这就是为什么说在写标准的时候想要巧妙地说出“vector 的存储模式必须是连续存储”（跟内建数组一样）这一要求是多么困难。在图 21-2 中，数组所占内存被认为是连续的，虽说数组中的各对象之间实际存在着“缝隙”（填充字节）。那么，“连续”的意义到底是什么呢？本质上，“连续”是指大小与结构大小相同的内存块之间是连续的，之所以标准中的定义是有效的，是因为“结构的大小”已经将填充字节算进去了。 158

C++标准保证所有通过 operator new 或 malloc 分配的内存对于你想要存储的任何类型的对象来说都是满足对齐条件的，这就意味着 operator new/malloc 必须遵从本地平台对于可能的数据类型的最为严格的对齐要求[4]。

---

1. 只有差劲的实现才会加入不必要的填充（即多于最小可能的填充）。
2. 不过，语言并不允许编译器擅自将示例 21-1 中的成员顺序调整成示例 21-2 中那样。标准要求所有位于同一个公用、保护或私有区段的数据成员必须保持它们声明的顺序。如果你的数据成员并非全部放在同一区段，编译器就可以通过调整各区段间的先后顺序来优化布局，这就是为什么有些人喜欢在每一个数据成员前都加上访问控制符的原因[3]。
3. 因为那样就等于允许编译器自由地调整每个数据成员的顺序了。——译者注
4. 例如在典型的 32 位机器上，最大的类型为 double，通常其大小为 8 字节。因此 operator new/malloc 分配的内存块的起始地址就是 8 的倍数。另外，一个结构的对齐要求就是该结构中最大的成员的对齐要求。——译者注

另外，正如前面描述过的，一个固定大小的分配策略可能会为特定大小的（某个基本大小 m 的倍数）内存块维护一个个的 arena[1]，而大小为 n 的内存申请则被圆整到 m 的下一个倍数。

内存与标准容器：基本问题

现在我们可以解决本条的最终问题了：

2. 给定同一类型同样数目的一组对象，各种标准库容器各消耗多少内存去存放它们？

不同的标准库容器使用了不同的底层内存结构，因而它们的“每对象”开销自然也就不尽相同：

- vector<T>内部是以连续的 C 风格数组来存放 T 类型对象的，因而根本不用为每个元素花任何额外开销（当然是除了用于内存对齐的填充字节之外；注意，这里的“连续”跟 C 风格数组所谓的连续是同一个意思，如图 21-2 所示）。
- deque<T>可被当作 vector<T>来看待，不同的是它将 vector<T>内部的存储空间拆成了一块一块的。deque<T>是一块一块（chunk）[或谓之一页一页（page）]地保存对象的。其每一页的实际大小在标准里并没有明确说明，主要依赖于存储的对象的大小以及你的标准库实现在大小方面作出的决策。这种分页式存储要求 deque 为每一页都保存一个额外的指针以便管理其相关信息，这整个的开销如果分摊到每个存储的对象的话就只有少之又少的一点点开销了；例如，在一个 8 位为 1 字节、4 字节为 1 整型/指针的系统上，一个页大小为 4K 的 deque<int>中，每个 int 变量只划到了 0.03125 位（bit）的额外开销，也就是说只有 1/32 位。除此之外就没有其他需要每个对象来承担的开销了，因为 deque<T>并不为其中的每个对象存储任何额外的指针或其他信息。另外，标准中没有任何强制要求说 deque 的页一定得是 C 风格的数组，这只不过是常用的实现方式而已。
- list<T>是一个保存着 T 元素的双向链表。这意味着对于每个 T 元素，list<T>都要额外存储两个指针，一个指向前一节点，另一个则指向后继节点。此外每次我们往 list<T>中插入一个 T 元素的时候，我们也会创建出两个新的指针，因此 list<T>中的每个元素至少需要花费两个指针的额外开销。

159

- set<T>（还有 multiset<T>、map<Key,T>或 multi-map<Key,T>）同样保存了持有 T 对象（或 pair<const Key,T>）的节点。set 通常的内部实现是树，其中每个节点都有三个指针的额外开销。人们看到这里往往会想：“为什么是三个指针呢？两个还不够吗？一个指向左子节点，另一个则指向右子节点。”这里之所以需要三个指针是因为我们同样需要一个“向上”的指针回指向父节点；否则从任意一个迭代器推导出其“下一个”元素就不可能足够高效地完成了。（除了用树之外，set 还有其他可行的内部实现方案，例如可以使用跳表，不过 set 中的每个元素仍然需要至少三个指针的额外开销。参见[Marrie00]中的一个例子。）

表 21-1 总结了每个容器的基本的额外开销。注意，有时候如果你愿意付出更高的“每迭代

1. 通常是自由内存块的链表，详见上文译注。

器”开销的话，你就可以得到更低的“每元素”开销。换句话说，有些低层细节信息是可以放在迭代器中的，从而增大每个迭代器的开销，换取更小的容器开销。不过到目前为止我还没听说过哪个商业实现使用了该做法。

表 21-1　各种不同的标准库容器所包含的每个元素的基本额外开销

容　器	典型情况下每包含对象所耗用的系统维护开销
vector	无
deque	几乎无（N 分之一比特）
list	两个指针
set,multiset	三个指针
map,multimap	三个指针（map 的元素类型为 pair<const Key,T>）

内存和标准库容器：现实世界

现在事情开始变得有趣起来：不要从表 21-1 中过快地**得出结论**。例如，仅从 list 和 set 所需的系统维护信息来看，你可能会认为 list 比 set 具有更小的每元素开销，毕竟 list 的每个元素**只需两个**额外的指针，而 set 却是三个。然而有趣的是，如果你将运行时内存分配策略也考虑进去的话，事情也许就不是这个样子的了。

再深入一层，考虑下面的表 21-2，它展示了 list、set/multiset、map/multimap 典型情况下使用的内部节点的布局。

表 21-2　各种不同标准库容器用于存放每元素的动态内存块

容　器	典型情况下每元素使用的内存块
vector	无，因为 vector 并不单独为每个对象分配内存块
deque	无，因为 deque 的所有对象都是分配在页中的，而几乎每页总是能存放很多个对象

list	struct LNode{
	LNode* prev;
	LNode* next;
	T object;
	};

set,multiset	struct SNode{
	SNode* prev;
	SNode* next;
	SNode* parent;
	T object;
	}; // 或等价的结构

map,multimap	struct MNode{
	MNode* prev;
	MNode* next;
	MNode* parent;
	std::pair<const Key,T> data;
	}; // 或等价的结构

接下来，考虑在如下的假设之下在实际中会发生什么事情（该假设是根据一个流行的平台而定的）：

- 指针跟 int 都是 4 字节长。（32 位平台的典型情况。）
- sizeof(string)是 16。注意，这只不过是 string 对象的“直接”大小（即 sizeof(string)），并没有将该 string 对象可能分配的任何数据缓冲区算进去；string 的内部缓冲区的数目跟大小是因实现而异的，但并不会影响下面所示的比较结果。（sizeof(string)是一个流行的 string 实现的实际结果。）
- 默认的内存分配策略是固定大小的分配，其分配的块大小为 16 字节的倍数。（微软 Visual C++的本地分配器的典型做法。）

160

表 21-3 中包含了一个简单的分析。你也可以自己尝试做这件事情：只需将数据调整为你自己的平台相关的数据，然后运用同样的分析方法。想知道如何编写一个能够计算出特定大小的内存分配请求的实际额外开销是多少的程序，你可以参考 Jon Bentley 的经典书籍最新版[Bentley00]中的附录 3。

观察表 21-3，我们立即就会发现一个有趣的现象：在许多情况下，或者说，在所有可能的元素大小中有 75%的可能情况下，list 和 set/multiset 实际上在我们假设的这个特定环境下的内存开销是一模一样的。这里还有一个甚至更为有趣的现象：list<char>和 set<int>在这个特定环境下的实际额外开销是一样的，虽说后者在其每个节点中存放了更多的对象数据和系统维护信息。

表 21-3 同样的每元素实际开销（实现相关的假设：sizeof(string)==16，指针和 int 皆为 4 字节大小，固定大小的分配块为 16 字节）

容器	节点内所含数据的实际总大小	为节点分配的内存块的实际大小，包含了对齐及块分配的额外开销
list<char>	9 字节	16 字节
set<char>, multiset<char>	13 字节	16 字节
list<int>	12 字节	16 字节
set<int>, multiset<int>	16 字节	16 字节
list<string>	24 字节	32 字节
set<string>, multiset<string>	28 字节	32 字节

如果内存耗用对你在特定场合下选择数据结构是一个重要考虑的话，请花几分钟作一下类似的分析，看看在你的当前环境中它们之间（的内存耗用）到底有哪些区别，有时它们所耗用的内存会比你想像的更少！

162

小结

每种容器都会选择不同的空间/性能平衡点。在效率方面，使用 vector 和 set 你可以做到 list

无法高效地做到的事情，譬如说 O(logN)时间复杂度的搜索[1]；你也可以利用 vector 做到 list 或 set 无法做到的事情，譬如随机的元素访问；你可以使用 list 来高效地完成某些任务，而 set 在完成这些任务时可能就稍逊一筹了，vector 则可能更慢，譬如说在中部插入元素等等。为了得到更好的灵活性，我们通常得花更多的内存开销，不过如果你将数据对齐和内存分配策略都考虑进去的话，这些容器之间的实际开销差别可能会远远出乎你的意料！你也可以参考 *Exceptional C++* [Sutter00]的第 26 条中有关数据对齐和空间优化的相关讨论。

准则：弄清你究竟得到了什么：弄清动态内存分配以及不同容器的实际的（和与系统相关的）空间开销。

163

1. 前提是 vector 的所有元素是有序的。

第 22 条 进行 new 操作，也许会抛出异常之一：new 的方方面面

难度：4

任何类只要提供了自己的operator new或operator new[]，就得同时提供对应的类相关版本的简单new（plain new）、定位new（in-place new）以及nothrow new。不这么做可能会给类的用户带来不必要的问题。

JG问题

1. C++标准中提供了哪三种形式的new?

Guru问题

2. 什么是“类相关”的 new，如何使用这种 new？描述在提供自己的类相关 new 和 delete 时任何需要特别注意的地方。

3. 在如下的代码中，从标注为 1 到标注为 4 的代码行分别调用了哪种 operator new?

```
class Base {
public:
  static void* operator new(std::size_t, const FastMemory&);
};

class Derived : public Base {
  // ……
};

Derived* p1 = new Derived;                          // 1

Derived* p2 = new (std::nothrow) Derived;           // 2

void* p3 = /* 足够放得下一个Derived对象的有效内存 */;
new (p3) Derived;                                   // 3

FastMemory f;
Derived* p4 = new (f) Derived;                      // 4
```

164

解决方案

在本条及下一条中，我想要陈述并证明如下两个主要的建议是正确的：

- 任何类只要提供了自己的 operator new 或 operator new[]，就得同时提供对应的类相关版本的简单 new、定位 new 以及 nothrow new。不这么做可能会给类的用户带来不必要的

麻烦。

- 避免使用 new(nothrow)，当你在对一次 new 操作失败进行检查时，确保你真的是在检查你所认为你正在检查的。

以上建议听起来可能有点儿令人诧异，那么就让我们来考察一下其背后的原因和基本原理吧。本条款主要讨论第一条建议，第二条建议则留待下一条款讨论。为了简单起见，我就不打算特地讨论数组形式的 new 了；文中对 new 的单一对象形式所作的讨论对其数组形式照样适用。

定位new、nothrow new以及简单new（即我们最常用的那种new）

1. C++标准中提供了哪三种形式的 new?

C++标准提供了三种形式的 new，并允许人们对其进行任意数目的重载。

其中一个有用的形式是定位 new，定位 new 负责在给定的内存地址处构造起一个对象来，此举不会分配任何新空间。例如：

```
// 示例22-1(a): 使用定位new，相当于一次“显式的构造函数调用”
//
void* p = ::operator new(sizeof(T)); // 获取足够多的内存用于存放T对象
new (p) T;      // 在p所指之处构造一个T对象，可能调用
                // ::operator new(std::size_t, void*) throw()
```

C++标准同样提供了“简单的旧式 new”（plain old new），这种形式的 new 不接受任何额外参数；此外还有 nothrow new，这种形式接受额外的参数。下面是标准 C++中提供的 operator new 重载的一个完整罗列：

```
// 标准提供的operator new重载
// (此外还有对应的数组形式new[]):
//
void* ::operator new(std::size_t size) throw(std::bad_alloc);
                        // 最常用的简单new
                        // 用法: new T

void* ::operator new(std::size_t size, const std::nothrow_t&) throw();
                        // nothrow new 用法: new (std::nothrow) T

void* ::operator new(std::size_t size, void* ptr) throw();
                        // 定位new 用法: new (ptr) T
```

165

标准允许程序员替换除最后一种 new 之外的所有其他形式的 new，即用自己的版本来替换它们。所有这些标准函数都位于全局名字空间，而不是 std 名字空间当中。简而言之，表 22-1 总结了标准版 new 的主要特性：

表 22-1　标准版 new 的相互比较

标准版的 new	额外的形参	是否进行内存分配	是否可能失败[1]	抛出异常	是否可替换
简单 new	无	是	是(抛出)	std::bad_alloc	是
nothrow new	std::nothrow_t	是	是(返回 null)	无	是
定位 new	void*	否	否	无	否

下面是一个例子，展示了 new 的这些不同版本的使用方式：

```
// 示例22-1(b): 使用各种形式的new，包括标准中提供的new，
// 还有用户自定义的new
//
FastMemory f;
new (f) T;                    // 调用某种用户自定义的new:
                              // operator new(std::size_t, FastMemory&)
                              // (或相似的，允许实参类型转换),
                              // 一般出现这种形式的new基本都是从一块自定义的arena取内存

new (42, 3.14159, "xyzzy") T; // 调用用户自定义的new:
                              // operator new(std::size_t, int, double, const char*)
                              // (或相似的，允许实参类型转换)

new (std::nothrow) T;         // 可能调用的是标准的或用户自定义的new
                              // operator ::new(std::size_t,
                              // const std::nothrow_t&) throw()
```

166

从示例 22-1(a)加上 22-1(b)中的所有用法可以看出，调用 new 时放在括号内的额外实参对应于相应 operator new 声明时形参列表里多出来的形参。当然，跟 22-1(a)中的情况不同，22-1(b)中的例子可能确实以某种方式分配了内存，并非使用现有内存。

类相关的new

2. 什么是“类相关”的 new，如何使用这种 new？描述在提供自己的类相关 new 和 delete 时任何需要特别注意的地方。

除了允许程序员替换某些全局 operator new 之外，C++还允许类也具有自身的类相关的 new。在阅读示例 22-1(a)和 22-1(b)时，你是否注意到“可能”这类字眼出现在注释中？下面就重复这两处地方：

```
new (p) T;   // 在p所指之处构造一个T对象，可能调用
             // ::operator new(std::size_t, void*) throw()

new (std::nothrow) T; // 可能调用的是标准的或用户自定义的new
 // operator ::new(std::size_t,
// const std::nothrow_t&) throw()
```

1. 在 operator new 的调用返回之后，对象的构造函数会被调用，当然这个构造函数的动作也仍是有可能失败的，但这儿我们并不关心这个，而是分析 operator new 自身是否会失败。

为什么我只说“可能”呢？这是因为被调用的 new 不一定是位于全局名字空间中的，而可能是类相关的 new。为了弄清这一点，请注意类相关的 new 跟全局 new 之间的两个有趣的相互作用：

- 尽管你不能直接将标准的全局定位 new 替换掉，但你却可以编写自己的、类相关的定位 new，这么一来当你对拥有该类相关的定位 new 的类进行定位 new 时，调用的就是你自己定义的定位 new 而不是全局的那个了。
- 你可以在不替换全局 nothrow new 的情况下添加类相关的 nothrow new。

这就是说在上面的两行代码中，有可能 T（或 T 的基类）提供了自己的类相关版本的 new，从而代码中的一处或两处所调用的就有可能并非全局版本，而是用户自定义的版本了。

下面就是一个简单的示范类相关 new 的例子，其中的类 X 提供了自己的、类相关的 new，它们分别对应于三种不同风格的全局 new：

```
// 示例22-2: 样例: 类相关版本的new
//
class X {
public:
  static void* operator new(std::size_t) throw();                    // 1
  static void* operator new(std::size_t,
                            const std::nothrow_t&) throw();          // 2
  static void* operator new(std::size_t, void*) throw();             // 3
};

X* p1 = new X;                                                       // 调用1

X* p2 = new (std::nothrow) X;                                        // 调用2

void* p3 = /* 足够放下一个X的有效内存 */;
new (p3) X;                                                          // 调用3 (!)
```

167

我特地在“调用 3”后边加了一个感叹号，这是为了再次强调一下一个事实：虽说你不能替换掉全局的定位 new，但你可以提供自己的类相关的定位 new。

名字隐藏所导致的令人惊讶的行为

为什么我要先介绍上面这些机制呢？原因是为了引出下面的名字隐藏问题：

3. 在如下的代码中，从标注为 1 到标注为 4 的代码行分别调用了哪个 operator new？

```
// 示例22-3: 名字隐藏了的 “new”
//
class Base {
public:
  static void* operator new(std::size_t, const FastMemory&);
};
```

```
class Derived : public Base {
 // ……
};

 Derived* p1 = new Derived;                    // 1

 Derived* p2 = new (std::nothrow) Derived;     // 2

 void* p3 = /* 足够放的下一个Derived对象的有效内存 */;
 new (p3) Derived;                             // 3

FastMemory f;
 Derived* p4 = new (f) Derived;                // 4
```

名字隐藏对我们中的大多数人来说也许并不陌生，不过人们通常见到的也许并非 new 的名字隐藏，而是诸如派生类中的名字隐藏基类中的名字这类隐藏，不过值得注意的是，名字隐藏有时也会发生在 operator new 的身上。

168

我们来回顾一下名字隐藏机制：简而言之，编译器会从当前域开始查找（本例中是从 Derived 类的定义内），查找需要的名字（本例中为 operator new）；如果没有找到该名字，编译器就会持续往外围作用域查找（本例中为 Base 类的定义内，然后是全局名字空间）。一旦它发现某个作用域内包含有需要的名字（本例中是在 Base 的定义内发现的）就会停下来，并就该作用域内的名字进行决议，这便意味着再往外层去的作用域（本例中是全局名字空间）就不予考虑了，从而其中的同名函数就被隐藏了；取而代之的是，编译器会在当下的名字空间中找到的与所求名字同名的实体之间进行决议，即通过函数重载决议选出一个被实际调用的函数，最后，编译器检查被选中的函数的访问规则看它是否可被访问/调用。需要注意的是，即便在这个过程结束后编译器仍没有发现可用的重载版本，刚才提到的再往外层的作用域中的名字也仍然是不予考虑的；另外，即便选出的重载版本是不可访问的（从而不可被调用），也要忽略再外层作用域内的名字。这就是 C++中的名字隐藏机制。（更多讨论参见 *Exceptional C++* [Sutter00]的条款 30 中关于名字查找和名字隐藏的讨论。）

上面这段话意味着如果一个类 C，或者其任何基类包含了类相关的 operator new，那么不管该 operator new 具有什么样的签名，它都会隐藏所有的全局 new，这就导致你无法为 C 编写再正常不过的、使用全局 new 的 new 表达式了。下面就是以示例 22-3 为前提编写的用户代码，从中你会看到我所说的情况：

```
Derived* p1 = new Derived;                    // 错误：无可用的匹配

Derived* p2 = new (std::nothrow) Derived;     // 错误：无可用的匹配

void* p3 = /* 足够放下一个Derived对象的内存*/;
new (p3) Derived;                             // 错误：无可用的匹配

FastMemory f;
Derived* p4 = new (f) Derived;                // 调用Base::operator new()
```

但如果我们非要使用全局 new 又该怎么办呢？对于上边的例子来说，至少前两个调用我们是希望使用全局 new 的，但由于基类中的重载版本并非有效的匹配，因此我们得想出一个合理的办法来将全局 new 重新放到编译器的眼皮子底下。对此惟一合理的途径是让 Derived 自身提供必要的类相关的 operator new（并由它们来将调用转发给全局 new），否则调用代码就只能使用全局限定符来引导编译器选出全局 operator new 了。

这番讨论引出了一些有趣的结论，最好将它们提炼为编码和设计方针。（Scott Meyers 在他的[Meyers97]的条款 5 中提到了部分有关内容，但其他方面同样重要。）

准则：如果你为某个类提供了任何类相关的new，那么记得同样为它提供类相关的简单（无额外参数的）new。

169

类相关的 new 在大部分情况下应当总是遵循相应的全局 new 的语义，因此声明的时候应当加上 throw(std::bad_alloc)异常规格，并尽量通过全局版本的 new 来实现它们，除非你真的需要插入一些特殊的手动处理：

```
// 类相关的简单new应当尽量像这样来实现:
//
void* C::operator new(std::size_t s) throw(std::bad_alloc) {
  return ::operator new(s);
}
```

注意，这里你可能是在调用一个被替换后的全局 new，而不是标准默认的那个，但这通常是件好事：大多数情况下，替换全局 operator new 的目的是为了调试或堆监测相关的原因，因此让这种行为反映在类相关版本的 new 上也是有必要的。

如果你不这么做的话，你的类的客户代码就无法以通常动态分配对象的方式来分配你的类对象。

准则：如果你为某个类提供了任何类相关的new，那么记得同样为它提供类相关的定位new。

类相关的定位 new 应当总是遵照其全局版本的语义，因此将它声明为什么异常也不抛出，并通过全局的定位 new 来实现它：

```
// 类相关的定位new最好像这样来实现:
//
void* C::operator new(std::size_t s, void* p) throw() {
  return ::operator new(s, p);
}
```

如果不这么做，调用你的定位 new 的客户代码就有可能遇到意外的行为（甚至被破坏）。特别地，标准库容器实现通常使用定位 new 来构建其中的元素，并期望这种定位构造的行为能够像标准默认的定位 new 那样；毕竟定位 new 是 C++中显式调用构造函数的惟一办法。除非你

为你的类编写了定位 new，否则你可能连 std::vector<C>都无法使用。

准则：如果你为某个类提供了任何类相关的new，那么考虑是否同样提供类相关的nothrow new（因为有些客户代码可能的确想要调用它）；否则它就会被其他的类相关new隐藏了。

170

你可以使用全局 nothrow new 来实现类相关的 nothrow new：

```
// "选项A"：类相关的nothrow new的实现。
// 注重与全局nothrow new的一致性。
// 应当等效于"选项B"
//
void* C::operator new(std::size_t s, const std::nothrow_t& n) throw() {
  return ::operator new(s, n);
}
```

或者，为了确保与常规 new 的语义一致（全局的 nothrow new 也应当遵循此一致语义，不过如果有人将全局 nothrow new 替换为不遵循此语义的版本会怎么样呢？），你也可以根据其语义来实现它：

```
// "选项B"实现类相关的nothrow new。注重与对应的
// 类相关简单new的语义的一致性。
// 应当等效于"选项A"
//
void* C::operator new(std::size_t s, const std::nothrow_t&) throw() {
  try {
    return C::operator new(s);
  }
  catch(...) {
    return 0;
  }
}
```

有一点值得注意，即这里展示的类相关new无法通过using声明来实现，例如"using ::operator new;"，这行 using 声明如果说非得找一个地方放的话那就是放在类 C 的定义体中，但可惜的是放在那里是非法的；在一个类的定义内部，你只可以通过 using 声明来带入基类中的名字，而不能带入诸如全局名字或其他类中的名字。要求调用代码自己去添加 using 声明不但费事，而且甚至根本就行不通，因为我们或许根本就无权改动它们——有些调用代码位于对我们来说是只读的模块当中（例如第三方库），或者甚至位于 C++标准库当中（如果我们试图向标准库容器提供对定位 new 的访问的话）。

小结

如果你提供了任何类相关的 new，那么：

171

- 应该总是同时提供类相关的简单（无额外参数的）new。

- 应该总是同时提供类相关的定位 new。
- 应该考虑也提供类相关的 nothrow new（因为有些客户代码可能的确想要调用它）；否则它就会被其他的类相关 new 隐藏了。

在下一条中，我们将会更深入探讨 operator new 操作失败究竟意味着什么，如何才能最好地检测出并处理掉这类失败情况。随之我们会看到为什么说避免使用 new(nothrow)也是个好主意，此外，可能最令你惊讶的是，我们将会看到在某些实际的、被普遍使用的平台上，内存分配失败通常甚至并不以标准中指出（必须）的方式呈现出来！你可要坐稳了。

172

第 23 条　进行 new 操作，也许会抛出异常之二：内存管理中的实际问题　难度：5

避免使用new(nothrow)，并且确保当你在检查一次new操作失败时，你真的是在检查你所认为你正在检查的。

JG问题

1. 标准的 new 如遇内存不足的情况，有两种主要的报告方式。请解释。

Guru问题

2. 使用 nothrow 形式的 new 能够令我们的代码更异常安全吗？证明你的答案。

3. 在现实中，有时候试图去检测内存是否已耗尽实际是不可能或无用的，试描述这类场合，不管是标准 C++中的还是位于严格意义上的标准 C++之外的。

解决方案

在前一条中，我们曾给出了如下的编码方针，并对其进行了阐述和论证：

- 任何类只要提供了自己的、类相关的 operator new 或 operator new[]，就得同时提供相应的类相关版本的简单 new、定位 new 以及 nothrow new。否则可能会给类的用户带来不必要的麻烦。

本条我们就来深入探讨 operator new 操作的失败究竟意味着什么，以及如何最妥善地对它们进行检测和处理：

- 避免使用 new(nothrow)，当你在对一次 new 操作失败进行检查时，确保你真的是在检查你所认为你正在检查的。

该主题的第一部分介绍的建议可能会令你感到有点吃惊，但后面的部分会更令你吃惊，因为在某些实际的、被普遍使用的平台上，内存分配失败通常甚至并不以标准中指出（必须）的方式呈现出来！

173

同样，为了简单起见，我就不打算特地讨论数组形式的 new 了；文中对 new 的单一对象形式所作的讨论对其数组形式照样适用。

异常、错误以及new(nothrow)

首先是一个回顾。

1. 标准的 new 如遇内存不足的情况，有两种主要的报告方式。请解释。

大多数形式的 new 通过抛出 bad_alloc 异常来报告分配失败，nothrow new 则通过历史悠久的 malloc 方式来报告失败（即返回空指针）。这确保了 nothrow new 永远都不会抛出异常，正如它的名字所暗示的那样。

问题是这是否真的给我们带来了好处：有些人错误地认为 nothrow new 加强了异常安全性，因为它防止了某些异常的发生。那么请回答下面这个问题：nothrow new 到底是加强了一般情况下的异常安全性还是加强了特定情况下的异常安全性？

2. 使用 nothrow 形式的 new 能够令我们的代码更异常安全吗？证明你的答案。

（可能会令你感到惊讶的）答案是：不能。错误汇报和错误处理是两个完全直交的（无关的）问题。究竟使用哪种异常汇报机制或处理机制只不过是语法上的差异罢了。

抛出 bad_alloc 跟返回空指针其实是两个等价的错误报告机制。因此，检测并处理分配错误也只不过是在检查异常跟检查空指针之间进行选择而已。

对于那些反正会进行错误检查的调用方来说，这其间的差别只是语法上的。这意味着不管采用哪种报错机制，客户代码都可以实现完全等价的程序安全性和异常安全性，即检测错误的语法与安全性是无关的，因为差异性只不过是在语法上面，而且只不过会对调用方代码的结构稍有影响而已（例如，if(null){ HandleError(); throw MyOwnException(); }跟 catch(bad_alloc){ HandleError(); throw MyOwnException(); }）。不管采用哪种错误汇报机制都不能提供额外的信息或额外的内在安全性，因此两者都不会令程序更“安全”或“能够更正确”，当然，前提是错误处理代码是正确编写的。 174

然而，对那些并不检查错误的客户代码来说，这其间是否有什么差别呢？如果客户代码不检查错误的话，惟一的区别就在于最终引发的错误在方式上会有所差别，但在严重程度上并没有任何差别：一个未被捕获的 bad_alloc 异常会粗暴地结束程序（不管程序栈是否会展开至 main），而一个未被检查的空指针则会导致一次内存非法访问，当用户试图对一个空指针进行解引用时程序会立即崩溃。这两种失败方式的危害性都是相当大的，但前者尚有一些优势：在栈开解的过程中至少会尝试销毁一部分对象从而令它们持有的资源得到释放，而且如果其中有些对象是像 TextEditor 这样的类的对象的话（TextEditor 对象在被过早销毁的时候会自动保存恢复信息），那么一个精心编写的程序可以防止程序的状态被丢得一干二净。（告诫：当内存真的是被耗尽了的时候，要在不试图获取哪怕一丁点儿内存的情况下正确完成栈开解和退出可谓难上加难）。与此相比，因使用坏指针而引发的崩溃则要“不健康”得多。

由此我们得出如下的教训#1：

准则： 教训#1：避免使用nothrow new。

---

nothrow new对程序的正确性或异常安全性并没有本质上的好处，某些失败情况下它甚至还

及不上异常，因为它有时候会掩盖某些失败，而异常至少有机会通过栈开解来进行一定程度上的恢复。正如上一条款中指出的，如果一个类提供了它自己的new，但忘记了提供nothrow new的话，nothrow new就会被隐藏从而变成不可用的。大多数情况下，nothrow new没提供任何好处，因此基于以上原因，我们应当避免使用它。

不过极少数情况下 nothrow new 也能带来好处，我所能想到的有两种情况。第一种情况属于在如今看来已经是“陈年旧事”的那类情况：例如一些非常老旧的、编写于 20 世纪 90 年代中期之前的 C++应用代码会通过检查（简单）new 的返回值是否为 null 来确定它是否失败了（当时简单 new 的行为的确如此，通过返回 null 来报告失败，但没多久这一行为就被修改了），当你将这类代码移植到现代编译器上时，（为了将对代码的影响降至最低限度）你可以将其中的简单 new 全局替换为 nothrow new，这样它的客户代码就不需要作任何改变；不过话说回来，自从简单 new 的行为改变到现在已经过了许多年了，这类代码的使用正在锐减。

另一种情况是在时间关键（time-critical）的函数内或内层循环中大量使用 nothrow new，而且所使用的编译器比较差，在编译涉及异常处理的代码时会引入较高的开销，并且在这一特定情况下使用常规 new 跟 nothrow new 会导致该函数的性能出现明显的差别，这种情况下使用 nothrow new 就能够给代码带来性能提升。注意，当我说“明显的差别”时，我的意思是我们得编写一个至少完整包含了那段时间关键代码的测试用例（而不是一个只去调用一下 new 的玩具程序），并对使用普通 new 跟使用 nothrow new 的情况分别测出耗时。如果在这之后我们得出的结论是使用 nothrow new 的确能给这段时间关键的代码带来性能上的提升，我们可能就要考虑使用 new(nothrow)了，而且同时得考虑其他提高分配性能的途径，比如编写一个自定义的、使用固定大小分配策略或其他快速内存 arena 的 new（不过在这么做之前，请先阅读第 21 条）。

175

由此我们得出教训#2：

准则：教训#2：无论如何，检查new是否失败通常都没多大意义。

这种说法可能会令一些人大吃一惊。有些人可能会义正词严地问：“你怎么能建议我们不去检查 new 是否失败了呢？难道这种检查不重要？！错误检查是程序健壮性的基石呀！”是的，我承认一般情况下的确是这样，但可惜的是，由于内存分配的特殊性，对于 new 来说失败检查往往是无意义的，这跟其他类型的操作不同，后者的失败可能的确应当被检查并处理。因此我们触及到第三个问题：

当理论遇到实际

在现实中，有时候试图去检测内存是否已耗尽实际是不可能或无用的，试描述这类场合，不管是标准 C++中的还是位于严格意义上的标准 C++之外的。

为什么说检查 new 的失败与否并不像有些人想像得那么重要呢？下面就是原因：

（1）有些操作系统直到内存需要被实际使用到的时候才会提交（commit[1]）实际的物理或虚拟内存，在这些系统上检查 new 失败与否可能会是无用的。在一些操作系统上[2]，内存分配总是会成功。

“嘿，等等”，你可能会问：“分配怎么可能总会成功呢，难道当没有可用内存的时候也能成功吗？”原因是在这类系统上分配只不过是记录一下对内存的请求而已，在幕后实际并没有发生任何物理内存（或虚拟内存）的提交，直到它们被实际使用到的时候操作系统才不紧不慢地交出物理或虚拟内存。甚至就算“分配”的内存被实际使用到的时候，通常物理（或虚拟）内存的提交也是根据进程实际访问的页面来逐页提交的，因此实际发生的情况可能是：在一个分配的内存块当中，只有被访问到的那些页面被提交了。 176

注意，在这类系统上，如果 new 直接去使用操作系统的内存分配设施的话，那么 new 就总能成功，而后面的那些看起来再正常不过的操作（如 buf[100]= 'c'；）则可能会抛出异常或导致失败，甚而导致程序崩溃。从标准 C++的角度来看，这些行为都是不符合标准的，因为 C++标准要求如果 new 不能提交足够的内存，那么它就必须失败（而上述的 new 则不然），而像 buf[100]='c' 这样的代码则不应该抛出异常或失败（而上述的行为则不然）。

背景：为什么有些操作系统会进行这种惰式分配呢？其实在这一策略背后有着一个著名的、切合实际的理念：一个进程可能请求了若干内存，但它并不一定会立即需要用上全部这些内存，实际上它可能永远也不会去使用这些内存，或者不会立即去使用它们，于是在这期间操作系统就可以将那些本应提交给该进程的内存“暂借”给其他进程使用。既然一个进程并不一定会立即需要用到它所请求的内存，那我们为何要急着将它所请求的内存全部提交给它呢？所以说这一策略确实具有一些潜在的优点。

说到缺点，除了不太符合 C++标准之外，该策略的主要问题在于，一般情况下它难以保证程序的正确性，因为任何对已分配内存的访问都有可能招致程序崩溃。这可不是件好事。反之如果失败在分配阶段就发生的话，程序就可以及早知道没有足够的内存以供它完成某个操作，从而去采取一些应对措施，譬如尝试分配一块较小的缓冲区，或放弃该操作，或者至少它能够通过栈开解来试图完成一些清理善后工作。但如果我们没办法得知分配是否真的有效的话，后续的任何对“分配”的内存的读写操作都可能会导致程序崩溃，而且这种失败还是不可预知的，你可能一上去就会碰一鼻子灰，也有可能在访问到缓冲区的第 100 万个页面的时候才遇到不测。

初看上去，似乎要想防止这种情况的惟一途径是在分配了内存之后立即对其进行“地毯式”写（或读），从而强迫系统“交出”实际内存。例如：

```
// 示例23-1: 手动初始化: 故意去接触每一个字节。
//
```

1. 提交意味着实际分配物理内存，在这之前你可以 reserve（保留）内存，即在地址空间里保留一块地址区域。

——译者注

2. 在 Linux、AIX 以及其他一些操作系统的某些版本上，惰式提交（lazy-commit）是默认或可定制的特性。

```
char* p = new char[1000000000];
memset(p, 0, 1000000000);
```

177

而如果你分配的是 non-POD[1]类类型的对象或数组的话，编译器隐式生成的构造函数就会自动帮你“踏遍”这些内存：[2]

```
// 示例23-2: 默认初始化: 如果T是非POD类型的，那么下面的代码就会随即初始化所有的T对象
// 从而触及每一个（有意义的，非填充字节）对象字节。
//
T* p = new T[1000000000];
```

如果 T 是一个非 POD 类型，那么数组中的每个对象都会得到默认初始化，这便意味着每个对象中的有意义字节（非填充字节）都会被写入，因此它们所占的内存必须被访问到[3]。

你可能会觉得这种做法是有用的。但实际上并非如此。诚然，如果示例 23-1 中的 memset 或示例 23-2 中的 new 成功地完成了，那么就能肯定我们的内存不但得到了分配而且已被提交。但问题是倘若失败了呢？那么结果并不会像我们期望的那样——抛出 bad_alloc 异常或返回 null，而是会导致访问违规进而使进程崩溃，对此我们的代码无能为力（除非使用某些平台相关的手段来截获访问违规）。这比什么也不做只是期望分配的内存的确会“在那儿”要稍微好一点，但也好不到哪儿去。

于是我们又回到了原来的问题，即是否符合标准的问题。由于::operator new 和::operator new[]本身就是编译器提供的，所以作为编译器自然要比程序员更容易把这件事做得更好。特别地，编译器实现者或许能够利用对操作系统的认知来截获访问违规并进而阻止程序崩溃。换句话说，C++实现者可能可以做到如下事情：分配内存，然后通过写每个字节（或至少每个页面）来“确认”分配，并以平台相关的手段来捕获这期间发生的失败（通常是访问违规），然后将其转换为标准的 bad_alloc 异常（对于 nothrow new 来说则是 null 返回值）。不过我怀疑没有哪家编译器厂商会去惹这个麻烦，原因之一是性能开销，而且只要使用了 new 就得承受这笔开销；原因之二就是在现实中 new 失败的可能性相当小……于是引出我们的下一个论点：

178

（2）在现实中，new 失败的可能性本就相当小，而又因为系统颠簸的存在则变得几近绝迹了。实际上，许多现代的、基于服务器的程序基本从不会遇到内存耗尽的情况。

在一个虚拟内存系统上，大多数现实中的基于服务器的软件都是在内存的多个不同部分完成

1. POD 的意思是“plain old data”（简单的旧式数据）。POD 的不太严格的定义是：一种数据类型，其中含一组简单数据，为了使用上的方便，可以具有用户自定义的成员函数。而较为正式一点的说法则是，POD 类型是指不含用户自定义的构造函数（包括拷贝构造函数）、析构函数、复制赋值操作符，以及不含引用、指向成员的指针以及非 POD 类型的（非静态）数据成员的类或 union[2]。
2. 标准[C++03;9/4]中对 POD-struct 的定义则是“不含非 POD-struct、非 POD-union（或它们的数组）以及引用类型的非静态数据成员，不含复制赋值操作符和析构函数的聚集体（aggregate[C++03;8.5.1]）”，而对“聚集体”的定义则是“一个数组，或一个不含用户自定义的构造函数；不含受保护或私有的非静态数据成员；不含任何基类或虚函数的类”。——译者注
3. 这并不包括那些其构造函数并不对对象数据进行初始化的不合理类型。

工作的，其他进程的情况也一样（只不过是在它们各自的内存中）；这就导致了随着被使用的内存逐渐增多，页面调度也逐渐频繁，而通常进程永远也不会走到“new 失败”这一步。因为实际上还没等到内存完全耗尽，系统早就开始颠簸（thrash）了：随着虚拟内存页面不断在磁盘和内存中来回倒腾，系统性能将会变得越来越差，系统管理员于是开始杀进程了。

告诫：我之所以说“大多数”是因为人们可能创建出一个只知不断请求内存但并不怎么使用它们的程序。虽然这种情况并不常见，但这种可能性是存在的，至少我的经验告诉我是这样。当然以上论断对不支持虚拟内存的系统（譬如许多嵌入式系统和实时系统）来说并不适用，因为这类系统中有许多是如此的容不得错误（failure-intolerant），以至于它们甚至根本不使用任何形式的动态内存，更不用谈虚拟内存了。

（3）当你检测到 new 失败的时候并不总能有办法应付。正如 Andy Koenig 在他的“When Memory Runs Low”[Koenig96]一文中指出的，在大多数情况下，特别是在进行程序测试时，当 new 失败时默认让程序崩溃（通常至少会先尝试进行栈开解）实际上是最好的选择。

当然，有些情况下当 new 失败时你的确可以采取一些应付手段：例如，假设你想要记录一些诊断信息的话，new 的失败处理函数就是进行这件事情的绝佳地点。有时候我们可以采取保留一块“应急内存”缓冲区的策略；不过这么做的人应当清楚他们在做什么，并在他们的目标平台上仔细测试失败情况，因为这种做法并不一定会像人们所想的那样工作。最后，如果内存真的被耗尽了，你就根本无法依靠抛出一个非平凡的（例如非内建类型的）异常来汇报错误了；即便抛出像 throw string("failed")；这样的异常也不行，因为它通常会通过 new 来进行一次动态分配（具体要看你使用的 string 实现的优化程度了）。

是的，有时候你可以做一些事情来应付某些特定类型的 new 失败，但通常并不值得去这么做，还不如就让栈自动开解，让 new 失败处理函数（也许还包括一些日志记录）完成它的工作呢。

你应当检查什么？

的确，在某些特定情况下，检查内存是否耗尽并试图从中恢复的举动是有意义的。Koenig 在[Koenig96]中就曾给出了一些这方面的例子。例如，你可以在程序一开始就把你以后将会用到的所有内存一次性分配完成，然后自己去管理它们；如果你的程序注定是要崩溃的，那么它就会在所有事情开始之前崩溃掉。这种方法需要你进行一些额外的工作，而且要求你必须预先知道所有的内存需求。 179

我在产品代码中见到过的可恢复的 new 失败主要归结为这么一类情况：即分配的缓冲区大小是从外部输入中获取的。例如，考虑一个通信应用，其发送的每个包头部都保存了该包的大小，接受者收到包之后所做的第一件事情就是从它的头部读出包大小并分配一个足够大的缓冲区来存放包里面剩余的部分。在这种情况下，我就见过试图分配巨量缓冲区的情况，尤其是当数据流被破坏或编程错误导致表达包长度的字节混乱的时候。这种情况下，程序就应当对这类破坏进行检查（如果可能的话，最好是设计合适的协议来在第一时间阻止这类情况发生），一旦发现数据

无效或缓冲区大小无意义就中止操作，因为这种情况下程序通常可以继续做一些有意义的事情，例如减小包大小并重试传送，甚至也可以干脆放弃这一操作去继续干别的事情。毕竟，分配 2G 内存的失败并不代表真的“没内存用了”，可能还有 1G 内存呢！[1]。

还有一种特殊情况，在这种情况下 new 失败恢复也是有意义的：当你的程序乐观地试图分配一个巨大的工作缓冲区时，如果这时分配无法完成的话，你只需降低标准，即重试获取一个较小的缓冲区，直到得到一个合适大小的。这种做法的前提是假定程序整体上是能适应实际的缓冲区大小的，也能在有必要的时候进行大块内存的分配。

小结

避免使用 nothrow new，因为它如今并不能提供任何显著的优点，而且其特性通常要比简单 new（可能抛出异常）的还差。

180

记住，无论如何，检查 new 是否失败几乎是没什么意义的，原因有若干。

如果你理当关心内存耗尽的话，请确保你正在检查的是你所认为你正在检查的，因为：

- 在那些直到内存被用到时才去提交实际内存的系统之上，检查 new 失败通常是没有意义的。
- 在拥有虚拟内存的系统上，new 失败几乎不会发生，因为早在虚拟内存耗尽之前，系统通常就已经开始颠簸了，而此时系统管理员自然会杀掉一些进程。
- 除了一些特殊情况之外，通常即便你检测到了 new 失败，要是真的没内存剩下了的话，那么你也就做不了什么了。

181

1. 有趣的是，分配的缓冲区大小由外部指定是一个典型的安全弱点。恶意的用户或程序会尝试对此进行有针对性的攻击，尤其是试图导致缓冲区问题，这种做法是典型的也仍是最受欢迎的搞垮一个系统的途径之一。注意，试图通过分配失败来搞垮一个程序的做法被称为“拒绝服务”（dential-of-service）攻击，其意图并非是想获取系统的访问权（乃至控制权）；与此有关但又截然不同的固定长度缓冲区溢出（overrun-a-fixed-length-buffer）攻击在黑客社群中也是一个总能得到青睐的手段，而且更令人称奇的是仍然有那么多人在使用 strcpy 之类的不加检查的调用从而让他们的代码暴露在危险之中。

优化和效率

我们常常想让（或者需要让）程序的某部分运行得更快，而关于优化，有很多方面是值得探讨的。

人们常常听到这样的建议，说使用 const 关键字能够帮助编译器优化代码。这种说法正确吗？不正确吗？为什么？除了 const 之外，有些程序员喜欢借助于 inline 关键字来实施其他形式的优化。然而，inline 真的会影响一个程序的性能吗？如果是的话，请说出是在什么时候，以何种方式？不管是在程序员控制之下，还是其他情况下都行，内联能够在什么时机实施？

最后，我们以一则脑力游戏来结束该部分，它展示了当我们想要获得最佳优化效果的时候，领域相关的知识是何等的不可替代，而且我们也借此机会引出了一些低层的位操纵代码。

第24条 常量优化

难度：3

const正确性有助于编译器进行代码优化吗？一个典型程序员的反应是“应该会吧”。这就将我们带到了有趣的部分……

JG问题

1. 考虑如下的代码：

```
const Y& f(const X& x) {
  // ……对x进行处理并查找Y对象……
  return someY;
}
```

将形参和(或)返回值声明为 const 能够帮助编译器生成更优化的代码或改善其代码生成吗？为什么能或为什么不能？请解释。

Guru问题

2. 在一般意义上解释一下 const 的存在与否为何能或为何不能帮助编译器改善它所生成的代码。

3. 考虑如下的代码：

```
void f(const Z z) {
  // ……
}
```

问题：

(a) 在什么样的场合之下、当 Z 为什么样的类型时，以上代码中的 const 限定才能真正帮助编译器生成不同的、更好的代码？请讨论。

(b) 对于 (a) 当中提到的情况，我们讨论的是编译器优化还是其他类型的优化？请解释。

184 (c) 要想获得同样的效果还有什么更好的途径吗？

解决方案

const：并不像有些人想的那样只是一个小小的优化

1. 考虑如下的代码：

```
// 示例24-1
//
const Y& f(const X& x) {
  // ……处理 x 并查找 Y 对象……
  return someY;
}
```

将形参和(或)返回值声明为 const 能够帮助编译器生成更优化的代码或改善其代码生成吗？

简短地回答是“不”，它也许并不能。

为什么能或为什么不能？请解释。

在上例中，编译器怎样能够做得更好呢？能避免掉参数复制或返回值复制吗？不能，因为参数本来就是按引用传递的，而且返回值也是按引用返回的。能将 x 或 someY 的复制放在只读内存中吗？不能，因为不管是 x 还是 someY 其实都存活在该函数的作用域之外：x 是来自外界的，而 someY 则是传递给外界的。即便 someY 是 f 内部就地动态分配出来的，它及它的所有权也因为这次调用而转交给了调用方。

然而，对 f 函数体内部的代码存在哪些优化的可能呢？由于 const 的存在，编译器能够以某种途径来改善它为 f 的函数体生成的目标代码吗？由此我们引出了第二个，也是更为一般性的问题：

2. 大体上，解释一下 const 的存在与否为何能或为何不能帮助编译器改善它所生成的代码。

请回顾一下示例 24-1，当然，上面所提到的“形参的 const 性（constness）通常并不能帮助编译器完成某些优化工作”的论断在这里仍然是有效的：即便你调用的是一个 const 成员函数，编译器仍然无法假设目标对象 x 或 someY 的内存字节不会被改变[1]。此外还有其他问题（除非编译器进行全局优化）：编译器可能无法确切地知道其他地方是否存在着一个非 const 引用也绑定到了 x 或 someY 所引用的对象，也不知道这些非 const 引用会不会在 f 的执行期间一不小心被用到；编译器甚至不知道 x 和 someY 所引用的实际对象到底是不是被声明为 const 了。 185

之所以会出现上述那些其他问题，就是因为 x 和 someY 被声明为 const 并不代表它们所引用的对象物理上也是 const 的。为什么这么说？因为两者的类（X、Y）当中都可能具有 mutable 成员数据，另外，其成员函数中也可能会使用跟 mutable 关键字在本质上一致的 const_cast。实际上，f 本身之内的代码就可能会对 x 或 someY 使用 const_cast 或 C 风格的 cast，从而令它们的 const 限定符形同虚设。

不过有些情况下 const 倒的确能够起到一定作用，即在对象的定义点上使用 const 限定。这种情况下，编译器往往能够成功地将这些“真正的 const”对象放入只读内存，尤其当它们为 POD

1. 别忘了还有 mutable 成员，甚至 const_cast。——译者注

类型的对象时，因为这类对象的内存镜像往往可以直接在编译期创建[1]，因而可以直接放至程序的可执行镜像中。这种对象一般被称为“可放入 ROM 的”（ROM-able）。

const什么时候真正能够起到促进优化的作用

考虑如下代码:

```
// 示例24-3
//
void f(const Z z) {

 // ……

}
```

问题:

（a）在什么样的场合之下、当 Z 为什么样的类型时，以上代码中的 const 限定才能真正帮助编译器生成不同的、更好的代码？请讨论。

如果编译器知道 z 确实为 const 对象，那么即便它不作全局分析也可以进行一些有用的优化。例如，倘若 f 的函数体中包含了像 g(&z)这样的调用，编译器就可以确信 z 中的非 mutable 部分在 g 调用的过程中是不会被改变的[2]。

否则，示例 24-3 中的 const 对大多数 Z 类来说就都算不上是优化了，即便是优化也不是编译器代码生成方面的优化。

对于编译器代码生成来说，问题很大程度上归于编译器是否能够消除复制构造或将 z 放入只读内存区段。也就是说，如果我们知道 z 确实不会被 f 内部的代码改动的话就好办了，因为从理论上来说这就意味着我们或许能够直接使用调用方从外界传给该函数的对象，而不用对其进行复制（并使用这份副本）；或者我们也可以复制一份副本并将该副本放入只读内存（如果这恰巧能够提高速度或实际有此需求的话）。

186

一般来说，编译器是无法使用参数的常量性来消除复制构造或假定逐位（bitwise）常量性的。因为正如前面已经提到的，可能出差错的地方实在太多了。特别地，Z 可能具有 mutable 成员，或者有些人在有些地方（例如，在 f 函数体中、在其他函数中、或者在 Z 的某些直接或间接被调用的成员函数中）可能会进行 const_cast 或做些其他手脚。

但有一种情况，在该情况下编译器或许能够生成更好的代码，前提是:

- Z 的复制构造函数以及所有被 f 体内直接或间接调用到的成员函数的定义在 f 的定义处都是可见的；以及

1. 言下之意大概是指那些编译期常量，它们的值在编译期是已知的，因而可以直接将它们放入可执行文件的只读区段中。——译者注
2. 倘若 g 试图通过 const_cast 改变 z 的非 mutable 成员，那么结果就是未定义的。——译者注

- 上述的这些函数都相当简单，无副作用；以及
- 编译器在优化方面不那么保守。

这就意味着编译器能够通晓全盘，因而能够在“好像”（as if）规则的指导下消除复制构造动作（不改变语义），该规则即编译器可以不按标准所说的做，只要没改变符合标准的程序的行为即可。

作为几句题外话，有必要提一下：有些人可能会说：“还有一种编译器能够利用 const 生成更佳代码的情况：即当编译器进行全局优化时。”这句话当然本身并没什么错误，但问题在于假设我们将其中的“利用 const”删除掉，这句话仍然成立。更不用说目前全局优化仍然还不多见且开销非常昂贵了；这里的问题主要在于，全局优化利用了编译器对所用对象的全部可知信息，因而不管该对象是否被声明为 const，编译器都会作同样的优化，因为它是基于你实际做了什么（而不是基于你保证会做到哪些事情）来进行优化的，所以说你所做的是否与你保证的一样其实是无所谓的。如果你的代码反正是要经过全局优化的，那么在本例讨论的情况下 const 性明显并不能给编译器的优化工作带来多大帮助。

记得之前我说过：“示例 24-3 中的 const 对于大多数 Z 类来说就都算不上是优化了”以及“对于编译器代码生成来说”，然而，除此之外还有更多你连想都没想过的优化！而且 const 确实能够促进某些真正的优化：

（b）对于(a)当中提到的情况，我们讨论的是编译器优化还是其他类型的优化？请解释。

特别地，存在着一些实际的优化，例如 Z 的作者可以明智地选择特殊对待 const 对象（以获得优化的效果）。 187

作为一个示例，让我们假设例 24-3 中的 Z 类是一个 handle/body 类，例如一个使用引用计数来进行迟复制（lazy copying）的 String 类：

```
// 示例24-4
//
void f(const String s) {
  // ......
  s[4];          // 或使用迭代器
  // ......
}
```

既然在 const String 上调用 operator[]理应是不能改变该 string 的内容的，那么这种 String 类可能就会提供 operator[]的 const 重载版本并让它按值返回一个 char（而不是按引用返回一个 char&）：

```
class String {
  // ......
```

```
public:
  const char operator[](size_t) const;
  char&        operator[](size_t);

  // ……
}
```

类似地，String 同样也可以提供 const_iterator，后者的 operator*也按值返回一个 char（而不是 char&）。

如果 String 这么做了，并且如果你恰巧使用的是上述的 operator[]或迭代器，同时你又将函数的按值传递的参数声明为了 const，那么恭喜你！String 类就可以在无需你进一步帮助的情况下，通过避免深复制自动并完全地对你的代码进行优化了……

（c）要想获得同样的效果还有什么更好的途径吗？

……但你只需简单地通过引用来传递参数就可以获得完全相同的效果，甚至还有过之：

```
// 示例24-5: 很简单，只需换成引用——比示例24-3中的做法还好
//
void f(const Z& z) {

  // ……

}
```

188 而且这种做法的好处是，不管 Z 使用的是 handle/body 惯用法还是引用计数，都照样有效。

准则：避免按const值传递对象。尽量按const引用传递，除非它们是像int这样的复制开销很低的对象。

小结

人们普遍认为常量正确性能够帮助编译器生成更紧凑的代码。是的，const 的确是个好东西，但本条款旨在说明 const 主要是写给人看的，而不是编译器或优化器。

说到编写安全的代码，const 是个极好的工具，因为它能辅助编译器去对代码进行检查，从而令程序员编写的代码更为安全。即使是在优化方面，从它允许类的设计者更好地实现手工优化这个方面来说，它也是非常有用的；不过从帮助编译器生成更好的代码这个方面来说它就显得没
189 那么重要了，因为编译器本来就是通晓一切的。

第25条 再论内联

难度：7

快速回答：编译器在什么阶段实施内联？是否有可能编写一个能够保证从来都不会被内联的函数？本条款我们将会考虑内联的众多不同的实施时机，其中有些可能是你想都没想到的。最终的答案简单地说来就是："永远都不算太晚，没有什么是不可能的……"

JG问题

1. 什么是内联？

Guru问题

2. 编译器在什么（哪些）阶段实施内联？请选择：

（a）编码期

（b）编译期

（c）连接期

（d）应用程序安装时

（e）运行时

（f）其他时候

3. 附加题：什么样的函数是从来都不会被内联的？

解决方案

对于Guru问题里的第1个问题，你选择的答案是什么？如果你选择了(a)或(b)，那么你就不孤独了，因为像你这么想的人不在少数。这两者是最常见的答案，你可能会想起*More Exceptional C++* [Sutter02]中的条款8，其中我对C++的inline关键字作了一些详细的讨论。但如果那里的讨论已经是详尽周全了的话我就不用写本条款了。但事实并非如此，关于内联我们还有更多需要认识的地方。远不止我们在*More Exceptional C++*中讨论的那点东西。

写作本条款的原因是为了解释为什么问题2的准确答案是"任一或全部"，而附加题的答案则是"没有这样的函数"。想知道为什么吗？继续往下读。 190

简要的回顾：内联

1. 什么是内联？

如果你读过*More Exceptional C++* [Sutter02]的条款8的话，那么下面的几个小节对于你来说

只是一个回顾，你大可以略过它们，跳至关于答案(c)的讨论。

简单地说，“内联”意味着将一个函数调用“就地”展开（in-place）为被调用函数的函数体。例如，考虑如下代码：

```
// 示例25-1
//
double Square(double x) { return x * x; }

int main() {
  double d = Square(3.14159 * 2.71828);
}
```

对上面这段代码进行内联的结果（至少从概念上）像下面这样：

```
int main() {
  const double __temp = 3.14159 * 2.71828;
  double d = __temp * __temp;
}
```

这里，内联消除了执行函数调用的开销，也就是将参数压栈并让CPU跳转至目标函数的入口点（这会导致引用局部性的丧失[1]）所花费的开销。注意，这里的内联并不代表将Square当成宏，因为内联函数调用仍然是函数调用，其实参只被求值一次。而如果用宏来实现的话，实参就可能会被多次求值，例如宏#define SquareMacro(x) ((x)*(x))，这样如果你调用SquareMacro(3.14159*2.71828)的话，它实际会被扩展成(3.14159*2.71828)*(3.14159*2.71828)（这相当于进行了两次“π跟*e*相乘”的操作，而不像函数调用那样只做一次）。

另外还有一个值得一提的特殊情况：递归调用。当进行递归调用时，一个函数会直接或间接地调用它自身。尽管这类调用通常是无法内联的，但某些情况下编译器仍然能够对某些递归进行内联，这很大程度上就跟它们对某些循环进行部分解循环差不多。

另外你有没有注意到示例25-1其实展示了一种不用inline关键字的内联？我给出这个例子是有意图的。在考虑下面的这个核心问题的过程中我们会不断地回到这个有趣的问题上来：

191 2. 编译器在什么（哪些）阶段实施内联？请选择：

（a）编码期

（b）编译期

（c）连接期

（d）应用程序安装时

（e）运行时

1. 现代硬件体系结构一般都支持指令预取，跳转指令或 call 指令会导致指令缓冲区被清空，从而影响执行效率。
——译者注

（f）其他时候

3. 附加题：什么样的函数是从来都不会被内联的？

答案A：编码期

在编码期，开发者可以在代码中放置inline关键字。严格意义上来说（如果内联是指以函数体代码来替换消除函数调用的话）这其实算不上真正的内联，但由于在代码中放置inline关键字相当于选择内联发生的合适地点，因此我们将它看成是进行内联决策的最早时机[1]。

如果你忍不住想在代码中添加inline关键字的话，你得记住三件重要的事情：

- *若非必要，别这么做。*不成熟的优化没什么好处，在没有对实际情况进行分析之前尽量不要写 inline。参见 *More Exceptional C++* [Sutter02]的条款 8，或在 Google 上搜"premature site:www.gotw.ca"，你会看到更多关于不成熟的优化（尤其是过早内联）的讨论及警告。
- *inline 的惟一作用是向编译器发出"邀请"*。正如 *More Exceptional C++*[Sutter02]中的条款 8 所述，inline 关键字的作用只不过是给编译器一个提示，作为一个语言中的关键字，它使程序员能够和编译器"对话"（这同时也带来了坏处，见下文）。不过它全部的好处也就这些了：inline 关键字的有无并不会对一个 C++程序的语义产生任何影响。同样，inline 也不会影响标准语言的其他部分，因为将一个函数设为内联并不会改变你使用该函数的方式（例如，你仍然可以取得该函数的地址），而且符合标准的 C++代码也无法以编程的方式来侦测出给定的函数是否为内联的。 192
- *inline 关键字往往显得"粒度"不够*。我们将一个函数设为内联函数，但实际上内联却是发生在函数被调用的点上。这是一个重要的区别，它意味着同一个函数可以（而且通常应当）在一个地方被内联而在另一个地方却不被内联。使用 inline 是无法表达这一事实的，因为我们只可以用它来修饰函数本身，因而当我们将一个函数设为内联函数时，我们其实相当于在说"我们认为我们知道将该函数在它的任意调用点上内联是合适的"。这一预见性的描述很大程度上是不准确的。因此，尽管我们常常称"内联某某函数"，但要想说得更准确一点的话最好统一改成"内联某处函数调用"。

准则：避免写inline或试图进行其他优化，除非性能测试显示出有此必要。

答案B：编译期

在编译期，编译器会按照惯例进行示例 25-1 中描述的内联。

当我们试图通过将特定函数声明为内联函数来"诱使"编译器去对其进行内联时，编译器实际会怎么做呢？答案是不一定的。并非所有的编译器都吃这一套，即便你"巧言令色"也不一定

1. 有疑义的是，"在编码期"进行内联的另一种解释为开发者可以自由地通过手动复制源代码来实现就地展开。这跟通常的"内联"的意义相差甚远，因此不讨论。

有用。你的编译器（或其他工具，下面我们会提到）可能并不会按你的指示行事，例如，你的编译器可能会：

- 拒绝在调用点上对你声明为内联的函数进行内联。
- 对你并没有声明为内联的函数进行内联。
- 对一个函数的某些调用点进行内联，其他调用点则不予内联（不管这个函数是否已经被声明为内联的）。

再次注意，示例25-1中并没有使用inline关键字。我之所以这么做是有目的的，因为我想让你知道，即便不写inline关键字，内联也照样可能发生。实际上，即便你现在用的编译器对示例25-1中的函数调用进行了内联的话，也不要感到惊讶。因为编译器这么做根本不会对一个符合标准的程序的语义或行为产生任何影响，编译器的这种做法属于从你的角度考虑可以（而且往往应当）进行的完全合法的优化。

在决定对哪些函数调用进行内联，包括是否对某个函数的某些调用点进行内联而在其他调用点不予内联时，现代编译器往往比程序员的表现要好。为什么呢？最简单的原因是编译器通晓更多的上下文知识，因为它知道调用点的“真实”构造，即在其他优化（诸如解循环以及消除死分支[1]）完成之后为调用点实际生成的机器代码。例如，编译器或许能够检测出在某个特定的内部循环中内联某个函数调用会导致循环体膨胀至无法放入高速缓存（cache），进而导致执行效率降低，于是编译器可能会决定对这一特殊情况放弃内联，同时仍然对该函数的其他调用点进行内联。

193

答案C：连接期

接下来我们开始考察关于内联的一些更为有趣的、也更为现代的方面。

问题：函数可能在连接期被内联吗？答案是肯定的。这个问题触及到了本条款开头的那个“附加题”的核心：什么样的函数是从来都不会被内联的？我之所以提出这么个额外的问题是因为人们普遍相信有些函数是不能被内联的。特别地，若一个函数的定义体没被放在头文件中而是被放在了其他独立的模块当中的话，这类函数通常被认为是不可内联的。

那么就让我们来试一试尽量不让编译器进行内联吧。考虑示例25-1的一个细微的变形：

```
// 示例25-2: 故意给编译器找茬：将函数放在另一模块中，
// 并让编译器无法看到该函数的定义体。
//

//--- file main.cpp ---
//
double Square(double x);

int main() {
  double d = Square(3.14159 * 2.71828);
```

1. 如果一个代码分支实际不可到达（不会被执行），那么该分支就被称为死分支（dead branch）。——译者注

```
}

//— file square.obj (or .o) —
//
// 包含了以下函数的编译后的目标代码:
// double Square(double x) { return x * x; }
```

我们将 Square 的实现移出了 main.cpp 编译单元。实际上我们所做的并不止这些：Square 的源代码甚至也是不可用的，惟一可用的就是它的目标二进制代码。于是有人可能忍不住要说“这种情况下对 Square 的调用当然肯定是无法内联的！”的确，对“编译器”来说他们的这些话是正确的。在编译 main.cpp 时，即便一个异常聪明的编译器可能也无法窥视到 Square 的定义。

然而，一个聪明的连接器却可以，而且实际上一些流行的商业连接器的确是这么做的。好些编译器产品，包括HP的编译器都已支持这种跨模块的内联了；而流行的Microsoft Visual C++ 7.0（又称“.NET”）及其更高版本如今也已支持这一特性，你只需打开编译器选项开关/LTCG即可，LTCG的含义即“连接时代码生成”（link time code generation）。这种晚期内联的一个实际优势在于优化工具能够知道每个调用点上下文的更多信息，因而对于何时何地值得对某处调用进行内联能够给出更为明智的答案。 194

但是且慢，这还没完：你有没有注意到在示例25-2的描述中没有任何地方提到Square必须是由C++写的？这其实体现了编译后内联的第二个优势：它是语言中立的。Square可以是Fortran写的，也可以是Pascal写的，都没问题。连接器惟一需要注意的地方就是传参惯例，并记得将参数压栈代码、参数弹出代码以及跳转指令一起去掉。

但是……且慢，还有，远远不止这些。系好你的安全带，因为真正的旅途才刚刚开始。

答案D：应用程序安装时

现在，我们的程序编译连接完了，我们将它做成压缩包或安装包，刻入CD，并自豪地发售给我们的第一个客户。那么此时我们可以放心大胆地宣称进行内联的时机终于过去了。

真的吗？

不然！

尤其是在20世纪90年代中期之后，越来越多发行的应用程序跑在托管的运行时环境上。换句话说，不是被编译为与特定芯片（硬件平台）及特定操作系统的API调用相关的机器代码，而是编译为所谓的“字节码流”，后者再由运行在目标平台上的运行时系统编译或解释为本地目标代码，这一做法的好处是由于中间层的介入从而将一些CPU和OS相关的设施抽象掉了，消除了平台依赖性。常见的例子包括（但并不仅仅是）Java虚拟机（Java Virtual Machine, JVM）、.NET公共语言运行时（.NET Common Language Run-time, CLR）[1]等。当编译器针对的是这类“中间”

1. 以及 CLR 的姐妹版，如 Mono、DotGNU 以及 Rotor，它们同样实现了 ISO 公共语言基础结构（Common Language Infrastructure, CLI）标准，后者是 CLR 的一个子集。

195 平台时，编译器就会将原始C++源代码编译为所谓的字节码流（也称为虚拟机指令语言或IL），编译为字节码流的程序代表着一个以运行时系统的指令集编写的程序。

题外话：这些运行时系统中有些具有非常庞大的的指令集，支持包括类、继承以及虚函数在内的高级面向对象概念。针对这类目标平台的编译器可以（而且许多正是这么做的）将源程序在保留类和函数的完整结构的基础上编译为指令语言代码，在这期间编译器会自己进行一些优化，包括进行一些编译期早期的函数调用内联。如果编译器这么做了，那么一个源代码形式的C++函数就可以在某种程度上直接以一个具有同样签名的函数来表示，只不过后者使用的是指令集。当然，编译器并不一定要这么严格地进行一一对应式的编译，但即使它并不一定得这么做，如下的关于内联的结论也依然有效。

回到我们主题上来：这跟在应用程序安装时进行内联又有什么关系呢？呃，即便是在托管的环境中，最终真正被用户机器CPU执行的也只能是本地二进制指令。因此托管环境会负责将中间语言代码进一步转换为目标平台的本地机器指令。而这又往往是在应用程序第一次安装的时候进行的，彼时正如同所有的类似编译的过程那样，可以（而且常常会）进行进一步的优化。特别地，.NET的NGEN IL编译器会在应用程序安装时，当部分或全部的中间语言代码被转换为目标平台可执行的本地指令时进行一些内联。

再次提醒你注意一下，这些托管环境中有些是语言中立的，因此在应用程序安装时进行的包括内联在内的优化皆可以跨语言进行。例如，一个C#程序调用了一个短小的C++函数，最后（在应用程序安装时）这个调用被内联掉了。当你发现这种情况的时候可别太惊讶！

那么究竟什么时候进行内联才“为时已晚”呢？永远不要说“决不会”，因为就算是现在故事也还没结束……

答案E：运行时

好吧，够了：到了运行时我们总该是脱离了代码的优化时机了吧？

运行时还能进行内联优化听起来似乎是天方夜谭，但实际上要想实现这一点仍然是有路可循的。特别地，我想提一下“性能导向的优化”（profile-directed optimization）以及“试探式内联”（guarded inlining）。正如即时编译环境一样（见前一节和后一节），这要求在用户机器的运行时环境上有相关工具的支持。 196

性能导向的优化是指当应用程序实际运行时，被预先安插到可执行程序中的设施会负责记录和收集关于程序实际被使用的情况，尤其是像哪些函数被频繁地调用，是在什么情况下被调用的（例如当某个函数被调用时工作集与高速缓存相比较的大小）等等。从这些“探测器”传来的信息可以用于指导修改可执行文件的内存镜像，对选定的函数调用进行内联，从而基于实际的运行时性能测试来将应用程序调整为最适应其目标环境的状态。

试探式内联则从另一方面说明了运行时内联的手段可以有多“激进”，[Arnold00]和[JikesRVM]中描述了针对JVM目标平台的Jikes Research Virtual Machine（RVM）也就是Jalapeño动态优化编译器。除了其他特性之外，该编译器还能够将虚函数调用进行内联，其工作方式是假设虚函数调

用的接受端是某个给定的类型从而直接去调用该类型上的相应函数（这不仅避免了函数调用的开销，而且还避免了虚函数分发的开销）。现今大多数编译器已经能够例行公事地将某些特定的虚函数调用“非虚化”了（进而还可以选择进行内联），这类可被非虚化的虚函数调用就是指那些调用的目标类型可以静态确定的虚函数调用。但Jikes/Jalapeño系统的新颖之处就在于，即便某个虚函数调用的目标类型无法静态确定，它也能够猜测性地对该虚函数的调用进行非虚化和内联。然而由于它的猜测并不一定正确，因此编译器会在被调用函数的入口点安插一个“哨位”，该“哨位”的用途就在于抢在函数被实际执行之前检查目标对象的实际类型是否符合编译器的猜测；如果不符合的话，执行流就会回退到原来调用的地方，并改为进行普通的虚函数调用。

答案F：其他时候

最后我还要给出一个我能想到的在“其他时候”进行内联的例子，它跟上面的某些例子虽说有所相似之处，但区别也很大，大到我认为应当独立出一小节来描述它。

回顾一下，在答案(d)中我们考虑了在某些托管的运行环境（例如JVM或.NET CLR）上进行应用程序的安装时发生的内联。当然，敏锐的读者或许早就已经注意到当时我只不过提到在安装期发生的字节码向本地机器指令的转换，但其实还有另一个更为常见的阶段也会发生这种转换，即JIT阶段，JIT即所谓的“即时”（just-in-time）编译。

JIT背后的思想是即时编译，即在函数被使用之前即时地将它编译出来。这么做的优点之一是可以分摊将程序编译为本地代码的开销：传统的编译是一次性地将整个程序编译为本地代码，而JIT则是一点一点地将程序代码的各个不同部分逐步编译出来，JIT发生在某段代码被执行的前夕。正因为JIT的这个特点，所以它可能会导致程序在第一次运行的时候稍有些慢，此外还会降低优化的质量（因为JIT必须够快，所以无法承受将大量时间花在分析内联及其他优化的可能性上）。不过JIT编译器仍然可以进行像内联这样的优化，而且许多JIT编译器的确就是这么做的，但如果我们将同样的工作放到更早期的话则通常会获得更好的效果，例如放到应用程序安装时（见答案D），因为应用程序安装久一点一般是无所谓的，所以优化器就可以不用太介意CPU时钟周期耗用，可以在优化前的分析上多花一点时间。 197

准则：记住，内联可能发生在任何时候。

小结

正如所有的优化一样，内联优化由那些通晓生成代码和执行环境的工具来执行往往能够取得比由程序员来“执行”更好的效果。越是晚进行内联，内联的目的性就越明确，越具有针对性。

本条款我们讨论了内联函数，不过更精确一点说应当是对函数调用进行内联。毕竟，同一个函数可能在它的某些被调用点上被内联了而在其他调用点上则没被内联。而且由于即使是在初期编译完成之后仍然还存在着许多内联的好机会，因此同一个函数不但可在不同地点被内联，而且可在同一地点由不同工具来内联。

看来，除了inline关键字以外，内联还意味着更多的东西。 198

第 26 条 数据格式和效率之一：什么时候压缩是真正重要的

难度：4

你在选择高度压缩、高效利用内存的数据结构方面有多擅长？你在编写那些“摆弄”字节（bit-twiddling）的代码方面有多大能耐？本条以及下一条会给你足够的机会去实践这两项技能，因为我们将会考虑国际象棋游戏的高效表示，另外还有一个用于存放它们的BitBuffer缓冲区类。

背景：假设你知道关于国际象棋的基本知识。

JG问题

1. 对于同样类型同样数目的一组对象，用哪种标准容器来存储它们所消耗的内存最少：deque、list、set 还是 vector？请解释。

Guru问题

2. 你准备创建一个世界范围内的国际象棋服务器，该服务器上存储了人们在它上面玩过的每一局的信息。由于存储这些信息的数据库可能会变得非常庞大，故而你要使用尽可能少的字节来存放每局信息。对于这个问题，我们只考虑玩家实际的移动，忽略诸如玩家名字和评论之类的信息。

对于下面给出的每种数据大小规格，给出一种能够表示一局国际象棋游戏的数据格式，要求使用下列对应的数据量来存储半步移动（半步移动是指由游戏的一方发起的一次移动）。假设 8 位一字节：

（a）每半步耗用 128 字节。

（b）每半步耗用 32 字节。

（c）每半步最少耗用 4 字节，最多耗用 8 字节。

（d）每半步最少耗用 2 字节，最多耗用 4 字节。

199 （e）每半步耗用 12 位。

解决方案

1. 对于同样类型同样数目的一组对象，用哪种标准容器来存储它们所消耗的内存最少：deque、list、set 还是 vector？请解释。

请回顾第 20、21 条，其中讨论了各种标准库容器的底层内存耗用和结构。每种容器都是在

空间/性能之间寻求不同的平衡点：

- vector<T>在内部以数组形式来连续存放 T 对象，因此没有每元素开销。
- deque<T>某种程度上可以被当成 vector<T>来看待，不过它的内部存储不像 vector<T>那样是连续的，而是分割成了一些区段，其中每个区段都是连续的。deque<T>存储对象的所谓的“页面”或 chunk（大块内存）。这一特性就要求它必须为每个页面保存一个额外的指针来管理该页面，不过这点额外开销若是分摊到每个元素的话也就只剩下微不足道的 *N* 分之一位（bit）了。除此之外 deque<T>并没有其他的每元素开销，因为 deque<T>并不单独为每个 T 对象保存任何额外的指针或其他信息。
- list<T>是一个双向链表，其节点内存放着 T 元素。这便意味着对每个 T 元素来说，list<T>都要耗费两个额外的指针开销，这两个指针分别指向链表的前一个跟后一个节点。同样，每当我们往 list 中插入一个新的 T 元素时，就会伴随着两个新的指针的创建，因此 list<T>的每元素开销是两个指针。
- 最后，set<T>（从这个意义上来说还有 multiset<T>、map<Key,T>或 multimap<Key,T>）同样保存了持有 T（或 pair<const Key,T>）元素的节点。set 通常是用树来实现的，其中每个节点里面都有三个指针（译者注：分别指向父节点、左子节点和右子节点）。人们看到这里往往会问：“为什么是三个指针呢？两个不就够了吗，一个指向左子节点，另一个指向右子节点？”。原因是 set 必须能够高效地遍历整个结构，所以我们还需要一个“往上”的指针，指向父节点，否则从任一元素出发得到其下一元素的请求就不能高效地完成了。（除了树之外 set 还可能有其他内部实现：例如跳表，不过跳表仍然要求每元素至少三个指针的额外开销（参见[Marrie00]））。

为我们的数据选择一个高效的内存表示的第一步是选择一个合适的（即空间/时间上最为高效的）容器（当然，前提是该容器必须支持你所需要的功能）。但仅仅这一点无论如何是不够的：另一个非常重要的步骤是为存放在容器之内的数据选择一个高效的内存表示。这一问题触及到了本条款的实质。 200

数据表示的不同途径

下面要讨论的这个问题的关键在于说明同样的信息可能有 *N* 种不同的表示方式：

2. 你准备创建一个世界范围内的象棋服务器，该服务器上存储了人们在它上面玩过的每一局的信息。由于存储这些信息的数据库可能会变得非常庞大，故而你要使用尽可能少的字节来存放每局信息。对于这个问题，我们只考虑玩家实际的移动，忽略诸如玩家名字和评论之类的信息。

本条款的剩余部分使用了如下的标准术语及缩写：

K　王

Q　后

R 车

B 象

N 马

P 兵

行 棋盘上横向的一排格子称为“行”，一共 8 格，通常编号为 1 至 8。离白方最近的一行为 1，最远的一行为 8。

列 棋盘上纵向的一排格子称为“列”，一共 8 格，通常编号为 *a* 至 *h*。靠白方左手的一列为 *a*，右手的一列为 *h*。

问题是：

对于下面给出的每种数据大小规格，给出一种能够表示一局象棋游戏的数据格式，要求使用下列对应的数据量来存储半步移动（半步移动是指由游戏的一方发起的一次移动）。假设 8 位一字节：

（a）每半步耗用 128 字节。

“耗资”如此巨大的这种表示方式是假设程序已经知道当前所有棋子在棋盘上的位置（可以根据前几步的移动推测出来）并将移动之后的整个棋盘的新布局保存下来，每格使用两个字节。这么做其实是在模拟一种标准的联机符号表示，即：使用'W'（白）或'B'（黑）或'.'（无）来表示哪一方占据了某一给定的格，使用'K'、'Q'、'R'、'B'、'N'、'P'或'.'来表示占据给定格的棋子。

使用该策略，并按从 1 行至 8 行、每行从 *a* 列至 *h* 列的顺序来存储整个棋盘，那么半步的表示就像这样：

201

```
WRWNWBWQWKWBWNWRWPWPWP..WPWPWPWP..... WP.............
.................................BPBPBPBPBPBPBPBPBRBNBBBQBKBBBNBR
```

如果以上数据表示玩家走出的第一步的话，那么其含义就是“1.d4”（或“1.P-Q4”），我通常的开局方式就是这个。

（b）每半步耗用 32 字节。

（a）中的表示方式似乎有点过于浪费了，因为它是人能读懂的 ASCII 文本格式，而我们其实只需要一个机器能读懂的格式即可。毕竟，运行在象棋数据库上的软件会负责将位置信息显示给用户。

我们可以将每个方格的耗用降至 4 位，这样每半步移动只需耗用 32 字节就可以了（仍然是保存整个棋盘的新布局，该策略不变）：我们需要 3 位来编码不同种类的棋子（例如 0 代表“王”，1 代表“后”，2 代表“车”，3 代表“象”，4 代表“马”，5 代表“兵”，6 代表“无”，虽然并没有 8 种可能性，但至少得三位才足够对它们进行编码），第 4 位则用于编码方格的颜色（如果方格里面没有棋子的话这一位就可以忽略）。

使用这种策略，并按从 1 行至 8 行、每行从 *a* 列至 *h* 列保存整个棋盘信息的话，每半步的表示可能会消耗以下这么多字节：

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

（c）每半步最少耗用 4 字节，最多耗用 8 字节。

要想达到以上效果，我们可以使用传统风格的象棋符号来表示半步移动。

传统风格的“描述性”的象棋符号通过像 K3 和 QN8 这样的不定长记号来表示方格（而不是像 e3 或 b8 这样的双字符记号）。要以这种方式描述半步至少需要 4 个字符（例如 P-Q4），而且最多可能会达到 8 个字符（例如 RKN1-KB1 或 P-KB8(Q)）。注意，这种编码方式不需要额外的结束符或分隔符就可以无二义性地进行解码。

使用这一策略的话，半步的表示方式就可能像这样：

P-KB8(Q)

（d）每半步最少耗用 2 字节，最多耗用 4 字节。

要想实现这个目标，我们可以通过现代象棋符号来对半步移动进行编码。

现代象棋符号表示具有代数的简洁紧凑，任意半步都可以通过至少 2 个字符（例如 d4）至多 4 个字符（例如 Rgf1,gh=Q）来表示。 202

同样，这种表示法也不需要任何分隔符就可以进行无二义性解码[1]。

使用这一策略,半步移动的表示可以像这样：

gh=Q

（e）每半步耗用 12 位。

我们可以通过另一条不同的途径来获得更高的空间利用率：设想我们只保存被移动的棋子的原先所在的位置和移动后的位置。要想对棋盘上的一个方格位置进行编码的话需要 6 位，因为一共有 64 格，即 64 种可能性，因此要想对两个位置进行编码，以允许记录下起始位置和移动后的位置的话，我们就需要动用 12 位。这对通常的移动来说是足够了，但对“兵升变”这种特殊情况来说则是不够的。

但无论如何该策略比先前的策略已经好许多了。现在，让我们暂且将压缩问题放至一边，下一条款我们将会考虑如何创建一个辅助数据结构来保存上面提到的这些位信息，因为它们的大小并不“整”，甚至会跨字节边界。 203

1. 顺便提一下，这种表示法在编程之外的一个主要优势就在于它可以由人们快速地写在纸上（即使时间紧迫）。它将 8 字符的上限降至了 4 字符，另外从概念上来讲也更为简洁，这两点给用户（也称玩家）带来了极大的好处。

第 27 条 数据格式和效率之二：（甚至更少的）位操纵

难度：8

本条我们考虑一些更为紧凑的、更高效地利用内存的数据格式，并着手编写一些操纵位的代码。

Guru问题

1. 为了实现第 26 条中 2(e)的解决方案，假设你决定创建如下的类来管理位缓冲区。可移植地实现它，以便让它能够在所有符合 C++标准的编译器（不管是什么平台）上正确工作。

```
class BitBuffer {
public:
 // ……根据需要增加其他函数……

 // 添加p所指的位缓冲区头部的num个位。
 //
 void Append(unsigned char* p, size_t num);

 // 查询正被使用的位数目，初始为0。
 //
 size_t Size() const;

 // 从第start位开始获取num个位，并将结果存入dst所指的缓冲区中（从缓冲区的第一位开始存放）
 //
 void Get(size_t start, size_t num, unsigned char* dst) const;

private:
 // ……在这里增加一些细节实现……
};
```

2. 有可能按每半步耗用少于 12 位的内存使用量来保存一局棋吗？如果有可能的话，请说明怎么做。如果不可以，请解释原因。

204

解决方案

BitBuffer，二进制杀手

1. 为了实现第 26 条 2(e)的解决方案，假设你决定创建如下的类来管理位缓冲区，可移植地实现它，以便让它能够在所有符合 C++标准的编译器（不管是什么平台）上正确工作。

首先得注意，“假设每个字节为 8 位”只在前一条款有效，在这里并不适用。我们需要一个能在任何符合 C++标准的实现上都可以正确编译运行的解决方案，而不管它到底是运行在哪个底

层平台上的。

我们所要求的接口总的来说像这样：

```
class BitBuffer {
public:
  void Append(unsigned char* p, size_t num);

  size_t Size() const;

  void Get(size_t start, size_t num, unsigned char* dst) const;

  // ……
};
```

你可能想知道为什么 BitBuffer 的接口是以 unsigned char 来表示位缓冲区指针的。首先，标准 C++中并没有“位指针”这样的概念，因此这种做法也就不可能了。其次，C++标准保证了对无符号类型（包括 unsigned char、unsigned short、unsigned int 以及 unsigned long）的操作不会遭到编译器的“非议”（如“嗨！你没有初始化这个字节！”或者“嗨，那不是一个有效的值！”）。正如 Bjarne Stroustrup 在[Stroustrup00]中指出的：

无符号整型是用于将内存当成位数组来看待的理想选择。

所以编译器被要求将 unsigned char（以及其他无符号整型）当成原始的存储位来看待，而这正是我们所需要的。除此之外还有其他途径，不过这一选择恰恰适合于用来实践我们的“位操纵”编码技术，后者恰巧是本条款的主要目的之一。

在实现 BitBuffer 时遇到的一个主要问题是：应当采用什么样的内部表示？下面我们就来考虑两个主要的方案。

方案#1：使用unsigned char缓冲区

第一个理想的选择是在内部用一个 unsigned char 数组来实现 BitBuffer，当我们将位放进该数组或从中取出来时，我们自己操纵这些位。我们可以让 BitBuffer 具有一个 unsigned char*类型的成员，它指向内部分配的 unsigned char 数组缓冲区，不过请稍等，请至少考虑一下使用 vector<unsigned char>吧，这样我们就无需担心基本的内存管理了。 205

你觉得以上这些听起来很简单吗？如果你是这么想的，而且你还没有试过去实现（并测试！）它的话，我建议你花上两三个小时去尝试一下。我打赌你试了之后就会发现，它并不像你想像得那么容易。

为了编写这个版本我耗费了相当多的心力，对于这一点我倒不觉得有多惭愧。仅仅是起草代码的最初版本所耗费的心力就大出乎我的预料，之后为了找出并纠正 bug 我又花了许多时间去进行调试。我并没有详细记录开发所耗的时间，不过回忆起来大约有几十次编译，其中有些是为了往代码中添加一些调试输出语句来输出某些中间值以供我分析并找出错误所在，另外还有 6、7

次调试是为了找出并纠正错误的。

下面就是我努力的成果。我不敢说它是完美的，不过它通过了我的单元测试，包括单字节和多字节添加以及一些边界条件。（你总是为自己的代码编写单元测试的，对吗？而且在你将代码 check in 到 CVS 里之前，你还会确保代码通过了你的单元测试，对不对？）注意，下面这个版本的 BitBuffer 是对大块字节缓冲区进行操作的。例如，假设我们使用的是 8 位的字节且当前写入偏移为 3，那么我们就会将待写入字节的前 3 位当成一个单元来整体写入，而将剩下 5 位当成一个单元来写入，也就是说每个字节分两步进行写入。为了简单起见，我还要求用户提供的缓冲区比必要的大小多出一个字节，因为这么一来我的代码就可以在缓冲区末尾后多余的一个字节上动点手脚，从而简化了代码。

```
// 示例27-1: BitBuffer是由vector<unsigned char>来实现的。
// 艰难而严峻的任务，唉！
//
class BitBuffer {
public:
 BitBuffer() : buf_(0), size_(0) { }

 // 添加p所指的位缓冲区头部的num个位。
 //
 void Append(unsigned char* p, size_t num) {
  int bits = numeric_limits<unsigned char>::digits;

  // 第一个目标字节和该字节中的位偏移。
  int dst = size_ / bits;
  int off = size_ % bits;

 // 不断push直到buf_足够大。——译者注
  while(buf_.size() < (size_+num) / bits + 1)
   buf_.push_back(0); // 鉴于下面的算法，这里的push_back必须为0。——译者注

  for(int i = 0; i < (num+bits-1)/bits; ++i) {
   unsigned char mask = FirstBits(num - bits*i);
  buf_[dst+i] |= (*(p+i) & mask) >> off; // 把当前字节的前bits-off位写入目标字节剩下的bits-off位
                                          //空间中。——译者注
  if(off > 0)
    buf_[dst+i+1] = (*(p+i) & mask) << (bits - off); // 把当前字节的后off位写入目标字节下一字节
                                                     // 的前off个空位中。——译者注
 }

 size_ += num;
}

// 查询正被使用的位数目（初始为0）。
//
size_t Size() const {
```

206

```
    return size_;
  }

  // 从第start位（位的编号从0开始）开始获取num个位，并将结果存入dst所指的缓冲区中（从缓冲区的第一位开始存放）
  // dst所指的缓冲区除了能放下num位之外至少还要多出一个字节。
  //
  void Get(size_t start, size_t num, unsigned char* dst) const {
    int bits = numeric_limits<unsigned char>::digits;

    // 第一个源字节和该字节中的位偏移。
    int src = start / bits;
    int off = start % bits;

    for(int i = 0; i < (num+bits-1)/bits; ++i) {
      *(dst+i) = buf_[src+i] << off;
      if(off > 0)
        *(dst+i) |= buf_[src+i+1] >> (bits - off);
    }
  }

  private:
    vector<unsigned char> buf_;
    size_t size_; // 位数

    // 创建一个前n位为1，剩余位为0的掩码。
    //
    unsigned char FirstBits(size_t n) {
      int num = min(n, numeric_limits<unsigned char>::digits);
      unsigned char b = 0;
      while(num-- > 0)
        b = (b >> 1) | (1 << (numeric_limits<unsigned char>::digits-1));
      return b;
    }
  };
```

上面的代码很重要。花点时间去阅读它并确认它所做的的确是对的。（如果你觉得你发现了 bug 的话，首先编写一个测试来演示这个 bug；一旦 bug 得到了确认，请将 bug 报告以及触发 bug 的测试用例一并发送给我。）

207

方案#2：复用标准的位容器

第二个主意是考虑到标准库中已经包含了两个能够存放位的容器：bitset 和 vector<bool>。不过相对于我们的要求来说 bitset<N>是个糟糕的选择，原因很简单：其长度是确定的（N），而我们要编码的却是长度可变的位流。因此没有选择的余地，vector<bool>尽管有着其他种种缺点，但毕竟是个诱人的选择，对于本例来说似乎是为我们量身定做的。（当然，标准实际上并没有要求 vector<bool>一定得使用紧凑的存储，即用一位来表示一个 bool 值，标准只不过是建议这么做；

但大多数实现的确正是这么做的。）

对于下面将要给出的这段代码，我所能说的最重要的一点是：

> 我在编写它们的时候第一时间就把事情做对了（而不像前一个版本的 BitBuffer 那样耗了许多心力）。

是的，没错。在我进行第一次编译直到代码最终确定之间，我所做的所有的工作就是纠正一些键入导致的语法错误，其中包括添加一个忘记打上去的分号（毕竟，找出这些错误本来正是编译器应该做的），在两处地方添加括号（因为我忘记了%的优先级比+高）。就这些。

```
// 示例27-2: 基于vector<bool>实现的BitBuffer。
//
class BitBuffer {
public:
 // 添加p所指缓冲区头部的num个位。
 //
 void Append(unsigned char* p, size_t num) {
  int bits = numeric_limits<unsigned char>::digits;
  for(int i = 0; i < num; ++i) {
   buf_.push_back(*p & (1 << (bits-1 - i%bits)));
   if((i+1) % bits == 0)
     ++p;
   }
 }

// 查询使用中的位数（初始为零）。
//
size_t Size() const {
 return buf_.size();
}

// 从第start位（位的编号从0开始）开始获取num位，并将结果存入dst所指的缓冲区中（从缓冲区
// 的第一位开始存放）。
 void Get(size_t start, size_t num, unsigned char* dst) const {
  int bits = numeric_limits<unsigned char>::digits;

  *dst = 0;
  for(int i = 0; i < num; ++i) {
   *dst |= unsigned char(buf_[start+i]) << (bits-1 - i%bits);
   if((i+1) % bits == 0)
     *++dst = 0;

  }
 }

private:
 vector<bool> buf_;
};
```

208

编写这个版本比示例 27-1 中的版本容易多了，这一点应当没什么可奇怪的。该版本复用了现有的位操纵代码，从而将代码量缩减了近一半，进而使得其中的 bug 大幅度减少。另外代码也变得更为清晰：这次我甚至不必要求调用方提供一点额外的输出空间来让我的 Get 代码更简单（上一个版本就是这样的）。

我怀疑上面的两个解决方案，特别是第一个尚有改进的空间，其中或许存在着我还没查出来的 bug，而且或许有些我没发现的地方代码还可以再简化一点，不过我个人觉得代码已经相当接近最优化了，无论是在正确性方面还是代码风格方面。

进一步压缩

最后让我们再来看一下国际象棋游戏的压缩表示，看看我们是否可以再做点什么，进一步进行压缩。

2. 有可能按每半步耗用少于 12 位的内存使用量来保存一局棋吗？如果有可能的话，请说明怎么做。如果不可以，请解释原因。

有可能，但要是你想将这种做法编码实现的话，你就需要像 BitBuffer 这样的位操纵容器的帮助了。

例如，有如下三种可行的途径：

通过对目标位置和移动到目标位置的棋子进行编码，我们可以将内存使用量降至每半步 10 位。因为其中编码一个方格（目标位置）需要 6 位（跟以前一样）。而编码被移动的棋子则可以简单地通过给每个棋子进行编号来实现：例如，可以按照棋子在棋盘上出现的先后顺序来为它们进行编号（具体谁先谁后则取决于按照什么顺序来遍历棋盘上的 64 格，例如从第 1 行至第 8 行，每行从第 *a* 列至第 *h* 列）。由于棋盘上最多只有 16 颗棋子，所以只需 4 位就可以对它们进行编码，加上前面的用于编码方格的 6 位一共是 10 位。 209

还能做得更好吗？让我们来分析一下：可以对所有可能的目标位置（棋盘方格）进行编码[1]，但通常在一次合法的移动中只有很少一部分方格是可以移动到的，因此我们的编码肯定还有精简的空间。类似地，我们可以对所有可能移动到给定方格的棋子进行编码，然而可以肯定的是并非所有的棋子都能够移动到给定方格，实际上有些棋子甚至根本哪都不能去，因此我们可能进行了某些没必要的编码。例如我们可以将半步的编码的第二部分加以压缩[2]，由于最多有 16 种可能，所以我们最多需要 4 位来对它进行编码，但如果只有一个棋子能够移动到给定方格的话，我们就根本无需对其进行编码了。另一方面，在解码过程中的每一步我们又都能知道究竟有哪些棋子能够移动到给定方格，因此我们也就知道从输入流中取多少位来解码当前的半步了。

我们可以以如下的方式来实现最少耗用 0 位最多耗用 8 位的编码：首先对所有合法的移动进行排序；例如，可以像前面提到的那样按照遍历棋盘的顺序来对棋子进行编号，并对每个棋子可

1. 即未被占用的棋盘方格。——译者注
2. 即表示棋子的那一部分。——译者注

能移动到的目标方格进行编号（同样按照棋盘遍历的顺序）。接下来只需使用最少的位数来保存每步实际移动的编码即可；例如开局有 20 种合法的走法，那么我们就需要$(\log_2(20)) = 5$ 位的简单二进制数来对它进行编码。

结论是，最少只需 0 位就可以编码每半步棋。0 位编码出现在有且仅有一种走法时，也称被迫移动。但最坏的情况需要多少位呢？这直接跟下面的问题相关：在一个合法的当前局势下最多能有多少种合法的移动？就我目前所能推断出的，一共是 218 种，右面是对应的局势图：

210

—	—	—	Q	—	—	—	—	3Q4
—	Q	—	—	—	—	Q	—	1Q4Q1
—	—	—	—	Q	—	—	—	4Q3
—	—	Q	—	—	—	—	R	2Q4R
Q	—	—	—	—	Q	—	—	Q4Q2
—	—	—	Q	—	—	—	—	3Q4
—	Q	—	—	—	—	R	p	1Q4Rp
—	K	—	B	B	N	N	k	1K1BBNNk

在这种最坏的情况下，我们需要用 8 位来将一次移动编码为二进制数。平均算来大概一次典型的移动需要耗用 5 位：开局可能有 20 种走法，而一个典型的棋局最后阶段（假设赢家有一王一车两兵）可能会有 30 种合法的移动（如果兵准备升变的话），这两种情况皆需要 5 位来编码。

只需对此稍加考虑，就会发现这种编码方式应当相当接近于最优了，因为它为眼下的问题[“玩家走的是哪步棋？（玩家进行的是哪一步合法的移动）”] 给出了一个直接而准确的答案。在通过前面的移动得知当前局势的基础上，我们使用最小数目的位将任何可能的下一步移动表示成了简单的二进制数。

我们还能做得更好吗？当然，不过再往下进行“吃力”跟“讨好”就不成正比了。这是因为进一步的优化依赖于我们对问题更多的认知或者只保存部分位。为了说明我们怎样才能做得更好一点，下面举一个例子：开局有 20 种可能的走法，根据前面提到的策略我们需要用 5 位去存储它。但实际上，即使假设所有移动可能都是平等的，开局第一步理论上也只有 $\log_2(20)=4.3$ 位的实际信息量，更何况平均起来我们应当能够使用甚至更少的位，因为白方的两个最为常见的开局走法占了绝大部分可能。简而言之，如果我们能够对每步走法的相对概率有更多的认知（例如将一个能够猜测给定局势下较好或较可能的走法的确定性的象棋机器人程序嵌进我们的压缩引擎），那么我们就可以使用诸如哈夫曼编码和算术压缩之类的变长编码策略了，这些策略的中心思想在于使用较少的位来表示可能性较大的走法。这其实是在用时间换空间，使用领域相关的知识来获取更好的压缩性能。

小结

本条展示了领域相关的知识如何能给问题的解决带来一个显著的飞跃。

总的来说，即便并不知道在给定局势下哪些走法更具可能性，我们也能够实现用大约 50 个字节来存储一个典型的 40 步棋局（80 个半步）的目标。这样的耗费已经算是相当少了，只有通过运用领域相关的知识才能实现最优化的解决方案。

准则： **仅当你明确知道：（a）应当进行优化；（b）应当怎样优化时，才着手进行优化。另外，领域相关的专业知识是无可替代的。**

211

陷阱、缺陷和谜题

在我们进入最后一部分深入研究一些风格案例之前，让我们停下脚步，考虑一些“别样”的C++问题或一些令人迷惑的情形。

为什么C++跟大多数程序设计语言一样都会为关键字保留名字，从而导致你无法创建一个名为class的变量之类的情况？为什么一行看上去本该做某件事情的代码实际上却什么也没做，同时编译器甚至不会为它生成任何机器指令（就好像那行代码根本不存在似的）？为什么看上去彻底是错误的代码却实际上能够正确、合法而可靠地编译和运行？为什么说当你在数字的海洋中漂浮（floating）的时候，你最好仔细检查（double-check）你的工作[1]？

最后还有关于糟糕恶劣的宏手法的一些告诫。

1. 作者这里旨在一语双关，用float和double两个字眼说明第30条讨论的问题。——译者注

第 28 条 不是关键字的关键字（或者：另一种注释） 难度：3

所有关键字都是平等的（对语法分析器而言），但是其中有一些比其他的“更平等”。在本条中，我们将会看到为什么说关键字得是保留字——因为关键字重要且特殊。我们还会接触到两个对C++程序的语义完全没有任何影响的关键字。

JG问题

1. 为什么大多数程序设计语言都将关键字视为保留字（即在程序里面不能随便使用的单词）？

Guru问题

2. 在一段 C++代码中加入关键字“auto”会对其语义产生什么样的影响？

3. 在一段 C++代码中加入关键字“register”会对其语义产生什么样的影响？

解决方案

为什么需要关键字？

对 C++而言，将关键字保留给语言自身而不作他用（例如作为类型、函数或者变量的名字）是非常重要的。

1. 为什么大多数程序设计语言都将关键字视为保留字（即在程序里面不能随便使用的单词）？

因为如果没有这些保留字的话，我们很容易就能够写出一些具有不可判定的二义性从而无法正常编译的程序。请看示例 28-1(a)这段极其简单的条件判断代码：

```
// 示例28-1(a): 一段合法的C++程序。
//
int main() {
  if(true);                    // 1: OK
  if(42);                      // 2: OK
}
```

214

编号为 1 和 2 的行分别测试了一个条件；如果条件判定为真（在本例中，均为真），则执行空语句。

毫无疑问这并不是我们所见过的最激动人心的代码。我当然也真心希望那并非你是上周所写过的最激动人心的代码。可它的确是合法的 C++代码，而且也是我所能想出来的最简单的代码，它展示了如果我们允许 C++的关键字被用作标识符的话会带来什么问题。

现在，让我们考察下面这段刚刚由另一个世界到达我办公室的代码（它悄无声息地出现，并且带着淡淡的樟脑和硫磺混合的味道），在那个世界里，C++并不将关键字作为保留字，大家可以很开心地将关键字作为标识符使用。那么，现在让我们先不考虑编译器该如何对付这种情况，而是转而思考一个简单得多的问题：我们人类将会如何理解下面例 28-1(b)所示的代码？

```
// 示例28-1(b): 并非合法的C++代码，但如果它合法的话会怎么样呢?
//
class if {              // 让我们将这个类称为“if”(这当然不合法，)
public:                 // 但如果它合法的话会怎么样呢?
 if(bool) {}            // 3: 嗯……调用构造函数？？
};
```

编号为 3 的行是什么意思呢？“噢，很简单，”有人可能会说，“我们都知道在这个地方条件判断语句很明显是没有任何意义的，况且一个类型名（bool）本身也不能作为被测试的条件表达式，所以，很明显，这里应当是构造函数调用。嘿！这么看来，让用户们重用类似于 if 这样的名字也许并不是个太糟的主意？毕竟我们还是可以分辨出用户此时的想法的！”因此一些语言的设计者随即就沿着这条肮脏的小路走下去了……但事实说明这条路的确是条肮脏逼仄的路，因为你并不需要走多远就会遇见下面编号为 4 和 5 的行所展示的情况：

```
// 示例28-1(b), 续: 现在让我们回到当初的代码……
//
int main() {
 if(true);              // 4: 嗯……这是啥意思?
 if(42);                // 5: 这个呢?
}
```

在这段来自另一个世界的代码中，编号为 4 和 5 的行是什么意思？它们和我们在示例 28-1(a)中看到的已经熟悉并喜爱、老旧但一览无遗的条件判断语句一样吗？还是调用了类 if（该类凑巧有一个合适且可用的构造函数）的构造函数呢？如果是后一种情况的话，代码的意思就是创建两个无名的临时对象。在对这个问题作了几秒钟的思考之后，我希望你能快速意识到，就算是人脑，对于本例给出的这种特定情况也无法得到确切答案，更不用说更普遍一般的情况了。既然连人脑都没有答案，我们又怎么能奢望编译器有答案呢？ 215

“可是，慢着！”那些还想在泥泞小道强行继续前行的人也许会说，“我们可以为这种情况制定一条规则，使其能够正常工作！在编号为 4 和 5 的两行，创建一个临时对象然后再将其摧毁是无用的，所以我们可以武断地认为这种行为并不是程序员所期望的，进而我们就能判定这里一定就是一个简单的条件判断语句了。”我希望看到这里你会因对这种丑陋的伎俩产生一种惊恐、厌恶、震惊以及沮丧的复杂情绪而退缩。但让我们鼓起劲来继续向前，给出两点压倒性的分析，它们足以将这个丑陋的伎俩打入万劫不复的深渊。第一点：既然你前面可以说创建一个临时对象再将其销毁是在做无用功，那我也可以很容易地反驳说“if(true);”如果作为条件判断语句的话它也只是一个空操作，所以绝不会是程序员真正的意图，并进而认定这两条语句都应当作为创建临

时对象来理解。第二点：不管你选择哪条路最终都会陷入一种尴尬的境地，即编号为 4 和 5 的两行表达式的含义完全取决于在当前域中是否存在着一个名为“if”的类，该类的存在与否会使得这两行表达式的含义产生天壤之别。那样的话就太令人伤脑筋了。

需要对特定情况制定这种黑客式的不正当的规则这一事实本身就在向我们发出强烈的警告，警告你此处存在着一些严重的设计问题。而实际上，也确实如此！

当然，如果关键字不作特殊保留的话，“创造”出此类二义性代码的方法还多的是。下面的代码就展示了另一个简单的例子：

```
// 示例28-1(c): 非法的C++代码，但倘若它合法会怎样呢?
//
class SomeFunctor {
public:
  int operator()(bool) { return 42; }
};

SomeFunctor if;     // 让我们称这个变量为“if”(不合法，但倘若它合法会怎样呢？)

// 现在我们再回到那段代码……
int main() {
  if(true);           // 6: 嗯……这句是什么意思?
  if(42);             // 7: 这句呢?
}
```

这里，请再一次思考，编号为 6 和 7 的两行分别是什么意思？它们和我们在示例 28-1(a)中看到的已经熟悉并喜爱的条件判断语句一样吗？还是使用了一个名为 if 的变量（该变量碰巧重载了 operator()）？更有甚者，对于后一种情况，它所重载的 operator()接受的参数恰恰就是 bool 型的（兼容于 true 和 42 这两个实参）！那么，到底在什么样的情况下这两行代码分别代表 if.operator()(true);跟 if.operator(42);呢？我想事情已经很清楚了：这种情况下，作为人类的你我无论如何都想不出一个合理的判定规则来，而如果连人们自己都不知道答案，那怎么能指望他写出一个知道答案的编译器来呢？

216

很明显，C++（和其他语言一样）需要将一些名字的意义如铁板钉钉般固定下来。它需要将那些名字保留为语言自身之用，而这些名字我们就称之为保留字。

关键字

C++标准保留了 63 个名字作为关键字，见表 28-1。我们应当熟悉这些名字中的大部分乃至全部，因为我们日复一日地使用着它们。

除此之外还有 11 种操作符可以用单词形式来表达。例如，在一个条件表达式中，你可以用“and”代替“&&”。标准同样也保留了这些名字，因此你也不能将它们用作你自己的名字。见表 28-2。

表 28-1　标准 C++关键字

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	int	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	export	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	True	
delete	goto	reinterpret_cast	try	

表 28-2　更多的保留字

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

217

以上列出的保留字一共有 74 个。数清楚了吗？这 74 个特殊的单词你都不能挪作己用，例如你不能将它们作为类型、函数和变量的名字（参见[C++03, 2.11 节]）。

关键字：少数派

大部分关键字都有其实际用途。这个当然，不然要它们干嘛？

尽管如此，有一些关键字却并不完全如你所期望的那样工作。事实上，它们中的一些对你的程序语义完全没有任何影响，真的！我的意思是说，事实的确如此，一些关键字在语义上完全等同于空白或者是装饰性的注释。更具体地说，我能给出 3 个这样的名字：auto、register；另外，在很多方面而言 inline（见第 25 条）也算是一个这样的例子。（如果你想知道关键字 export 在理论和实践中是否有实际作用的话，本书对此也有专门的论述；见第 9、10 条。）

auto

首先来考虑可怜的 auto：

2. *在一段 C++代码中加入关键字“auto”会对其语义产生什么样的影响？*

简单回答就是：完全没有影响。auto 是一个完全多余的存储类别指示符。它仅能出现在代码块中对象的声明语句里，表示这些对象将会在函数或者代码块结束的时候自动销毁；然而，在所有 auto 能够出现的地方，就算 auto 没有出现，“局部变量‘自动’销毁”这一规则也仍是有效的，这就是造成它多余的原因。简单地说，auto 可有可无。

这下某些敏感的读过 C++标准的人会跳起来大声嚷道：“可那并不完全是标准上所说的啊！

为什么标准甚至在一个标注中说道 auto 只是近似等同于多余呢！”好吧，标准上是这么说的：

…auto 指示符几乎总是多余的，而且并不经常被使用；auto 的一个用处就是显式地区别声明语句和表达式语句。——[C++03] 7.1.1 节

是的，标准的确是这样说的，可是不，它并不正确。（我已经为更正这个非正式的注释而提交了一个缺陷报告。）为什么说它不正确呢？因为 C++在处理这种二义性时的规则是：如果一个东西可以被解释为一个声明，那么它就是一个声明；而 auto 的加入也并没有改变这一点。例如：

```
// 示例28-2: auto的加入并没有消除二义性。
//
int i;
int j;
int main() {
int(i);              // 声明了一个 i; 而不是对::i的引用。
auto int(j);         // 仍然是声明; 并非是对::j的引用。
 int f();            // 一个函数声明, 并非一个默认构造的int变量。
 auto int f();       // 仍然是一个函数声明, 只不过这种写法。
}                    // 在一些严格的编译器上可能无法通过编译。
```

218

更进一步的有关 C++中声明的二义性的讨论，可以参见第 29 条（同样可以参见[Meyers01]的条款 39）。归根结底，auto 在这种二义性情况下并不能用来消除二义性。

准则：绝不要用auto。它跟空白没什么区别。

题外话：将来 auto 可能会变得有意义。随着 C++委员会在下一代 C++标准（也就是俗称的 C++0x）方面的工作的进展，人们逐渐开始把目光放到 auto 的另一种可能的复用上，即用作“自动”类型推导，如果该特性进入 C++0x 的话，到那时我们就可以极大程度上简化某些变量声明，例如将下面这样的语句：

```
vector<SomeNamepace::SomeType>::const_iterator i = v.begin();
替换为:
auto i = v.begin(); // 现在虽然是非法的，但有可能在以后的C++扩展中就是合法的了。
```

register

关于 auto 已经说得够多了，那 register 又如何呢？我们问：

在一段 C++代码中加入关键字“register”会对其语义产生什么样的影响？

简单回答就是：在大部分现代的编译器上，根本没影响。

要想知道为什么，让我们来看看 C++标准中紧接着 auto 的注释说了些什么……：

register 指示符与 auto 指示符拥有相同的语义……

啊哈。依照我们已经讨论过的，这就应该意味着“没有语义”。看来事情不大妙。继续往下

读，标准接着说道：

……对实现是一个提示，提示被修饰的对象将会被非常频繁地使用。[注释：特定的实现可能会选择忽略该提示，而倘若你对该对象进行了取地址操作，则大多数实现都会忽略该提示。——注释结束]——[C++03] 7.1.1 节

register 关键字背后的想法是如果某些变量会被非常频繁地使用，那么尽可能将它们放到物理 CPU 寄存器中（进行使用）将会是非常有价值的做法，因为平常 CPU 得从相对（寄存器来说）较慢的高速缓存、甚至更慢的主存中读取数据，而一旦将数据存放在寄存器中的话，CPU 就可以以高得多的速度完成操作。

到目前为止一切都好，但我们还没考虑程序员呢。 219

作为程序员的你应该几乎从来都不需要用到 register。如今程序员想要自己去控制所生成代码的寄存器分配情况的话，无异于白费力气，因为就算是最好的程序员（也就是你）也不太可能得出最优的寄存器分配方案，使得自己的代码运行到最快。就算程序员非常清楚地了解运行代码的芯片的各种细节（这已经很难得），并且和编译器开发团队一样熟悉编译器的代码生成机制（这就更加不可能了），他也不可能比编译器更善于把对象分配给寄存器，因为代码生成器在看到代码并决定哪些部分能够因放到寄存器中而获益时，原始代码已经经过了重重转换了（例如内联、解循环、消除死分支、变量折叠等），而程序员根本就不知道这其间究竟发生了哪些转换。你其实不仅做不到和你的编译器一样好，而且根本就不应该考虑自己亲手来做这个事情，显然这种事应该交给自动化工具来做。

准则：不要用register（除非你明确地知道你的编译器会特殊对待带有register的代码）。对大多数编译器而言它跟空白也没什么区别。

小结

我们已经考察了为什么 C++将关键字作为保留字，并且进而研究了两个对 C++程序的语义毫无影响的关键字：auto 和 register。请不要使用它们；它们跟空白没什么区别，所以何必要多此一举呢？

不要用 auto。它跟空白没什么区别。

不要用 register（除非你明确地知道你的编译器会特殊对待带有 register 的代码）。对大多数编译器而言，它跟空白也没什么区别。 220

第 29 条　这是初始化吗

难度：3

很多人都知道这句名言："如果他们发起一场战争却没人去参加怎么办？"本条我们要考虑的就是一个类似的问题："如果我们想要初始化一个对象，而实际上什么也没有发生，怎么办？"

假设相关的标准头文件都已经被正确地包含了，而且也作了适当的 using 声明。

JG问题

1. 下面这段代码的功能是什么？

```
deque<string> coll1;

copy(istream_iterator<string>(cin),
    istream_iterator<string>(),
        back_inserter(coll1));
```

Guru问题

2. 下面这段代码的功能是什么？

```
deque<string> coll2(coll1.begin(), coll1.end());

deque<string> coll3(istream_iterator<string>(cin),istream_iterator<string>());
```

3. 我们需要做什么样的修改才能使代码如程序员所预期的那样工作？

解决方案

前奏：基本机制

1. 下面这段代码的功能是什么？

```
// 示例29-1(a)
//
deque<string> coll1;

copy(istream_iterator<string>(cin),
        istream_iterator<string>(),
        back_inserter(coll1));
```

221

这段代码声明了一个名为 coll1 的 deque<string>容器，并将其初始化为空。然后，将从标准输入流（cin）中读入的以空格分隔的一个个 string 推入容器，直到没有更多可用的输入为止。

示例 29-1(a)中的代码，等效于：

```
// 示例29-1(b): 等同于29-1(a)
```

```
//
deque<string> coll1;

istream_iterator<string> first(cin), last;
copy(first, last, back_inserter(coll1));
```

这里惟一的不同点就在于，示例 29-1(a)中的 istream_iterator 对象是即时创建出来的一个无名的临时对象，因此在 copy 函数调用结束后，这些临时对象就会被自动销毁。而在示例 29-1（b）中，istream_iterator 对象是具名变量，它们在 copy 函数调用完毕之后仍然存在，直到脱离示例 29-1(b)中代码段所在的作用域范围才会被销毁。

间奏：出问题了

2. 下面这段代码的功能是什么？

```
// 示例29-2(a): 声明另一个deque
//
deque<string> coll2(coll1.begin(), coll1.end());
```

这段代码声明了另一个 deque 容器，名为 coll2，并且使用传递给构造函数的初始化参数对其进行了初始化。代码中使用了 deque 的构造函数之一，这一构造函数接受的参数是一对迭代器，表示一个有待复制的区间。在这种情况下，我们的 coll2 被迭代器所表示的区间内的元素（也就是 coll1 中所有的元素）所填充。

示例 29-2(a)中展示的代码等价于如下代码：

```
// 示例29-2(b): 跟示例29-2(a)几乎一模一样
//
// 额外的步骤: 调用默认构造函数
deque<string> coll2;

// 使用push_back往其中添加元素
copy(coll1.begin(), coll1.end(), back_inserter(coll2));
```

这段代码跟示例 29-2(a)的一个微小的差异就是 coll2 的默认构造函数首先被调用，然后，作为一个单独的步骤，利用 push_back 方法将 coll1 中的元素推入其中。而原来的代码则是简单地使用接受一对迭代器为参数的构造函数来完成同样的工作。 222

你可能会奇怪，为什么我在这里要不厌其烦地分析这个语法。请先不要着急，看完下列的最后一部分代码你就会知道原因。下面这段代码从某种意义上来说是完全无害的，而“无害”往往是好事，不是吗？然而不幸的是它比有些人所认为的或想要的要“无害”得多：

```
// 示例29-2(c): 要声明另一个deque?
//
deque<string> coll3(istream_iterator<string>(cin), istream_iterator<string>());
```

乍一看，这段代码完成的事情和示例 29-1(a)是相同的：创建了一个 deque 容器，并从标准

输入流读入 string 来填充此容器。稍微有点不一样的地方是，这里它试图采用示例 29-2(a)中的语法形式（即使用接受一对迭代器为参数的构造函数）来完成工作。

以上代码中存在一个潜在问题和一个实际问题。潜在问题是 cin 将会被耗尽，因此后面代码不能再从其中读入输入，这很可能会是个逻辑问题。

更大的问题是这段代码实际上根本不做任何事情！为什么会这样呢？因为这里实际上并没有声明一个名为 coll3 的 deque<string>对象，而是声明了（往下阅读之前，先深呼吸）：

一个名为 coll3 的函数
　会按值返回一个 deque<string>，
　　该函数有两个参数：
　　一个是类型为 istream_iterator<string>名为 cin 的形参，
　　另一个（无名）参数则是一个：
　　不接受任何参数，
　　并返回一个 istream_iterator<string>的函数。

（像不像绕口令？）

发生了什么事情？本质上，我们碰上了 C++为了保持与 C 的兼容性而从 C 那里继承来的令人痛苦的规则：如果一段代码能够被解释成声明，那它就是声明。用 C++标准的原话来说就是：

> *在声明和表达式语句这两者的语法形式之间可能会出现二义性：一个将函数风格的显式类型转换（_expr.type.conv_）作为其最左端的子表达式的表达式语句和一个其第一个声明子（declarator）以“(”开头的声明语句可能会无法区分开。在这种情况下，该语句被解释为声明。*
>
> ——[C++03] 6.8 节

223

大体上来说，之所以把规则制定成这样是为了帮助编译器对付 C 的那些讨厌的、可能存在二义性的声明语法，因此，为了控制事态，编译器在解决这类二义性的时候统一遵循这样的规则：即如果有二义性的话，就当它是一个函数声明。我想我说得够明白了。

如果你还未看过，那么现在请浏览一下《Exceptional C++》[Sutter00]的条款 42，其中讨论的示例和这里的是一样的，只不过更简单一些。那么，现在就让我们来一步步地仔细看看到底发生了什么事情：

```
// 示例29-2(d): 等同于29-2(c)，只不过删除了多余的括号，并添加了一个typedef
//
typedef istream_iterator<string> (Func)();

deque<string> coll3(istream_iterator<string> cin, Func);
```

这看上去是否更像一个函数声明了呢？好吧，可能你仍然不够肯定。那么就让我们再前进一步：去掉前面的形参名 cin（它反正也没什么用），同时将 coll3 改成一个我们习惯的函数名字：

```
// 示例29-2(e): 仍然等同于29-2(c)，只是名字上做了少许的改动
//
typedef istream_iterator<string> (Func)();
deque<string> f(istream_iterator<string>, Func);
```

现在代码清晰了许多：上面这段代码看上去已经非常像是一个函数声明了，因此按照 C 和 C++的语法规则，它实际上就是一个声明。之所以这个形式看起来很让人迷惑，是因为它也很像构造函数调用的语法；而那个形参的名字则让这种情况越发混乱：即 cin 看起来非常像是这个作用域内的一个变量名，而且这个名字甚至还是标准预先定义好的！（程序员的本意就是使用标准库定义好的 cin。）但是，最误导人的就是它们居然相安无事，因为形参名 cin 跟 std::cin 除了名字凑巧一样之外没有任何共同点。

在现实的编码工作中，人们仍然会一次又一次地遇到这个问题，这也是我为什么要在这里写这一条款的原因。实际上（可能会令人惊讶），这段代码仅仅是一个函数声明，并没有做任何实际的工作，即编译器不会为它生成任何实际的代码，没有任何操作会被执行，deque 的构造函数不会被调用起来，更不用说创建什么对象了。

224

正确的做法

当然，只顾提出问题而不告诉你如何解决它是不公平的。而这正是我们最后要讨论的。

我们需要作什么样的修改才能使代码如程序员所预期的那样工作？

我们所要做的就是避免让编译器将其识别为一个函数声明。有两个简单的方法可以做到这一点，最诱人的做法就是：

```
// 示例29-3(a): 通过添加括号来消除二义性(可行，得分：7/10)
//
deque<string> coll3((istream_iterator<string>(cin)), istream_iterator<string>());
```

这里我们只需给参数加上额外的括号，提示编译器我们这里提供的是构造函数的参数而非参数声明。这种做法之所以行得通是因为虽然 istream_iterator<string>(cin)可以被看成变量声明（当然，如上所言，也可以是参数声明），但(istream_iterator<string>(cin))却不行，它只能被解释为表达式，因此，示例 29-3(a)中的代码不可能是函数声明，这跟 void f((int i))不可能是函数声明是一回事，那对冗余的括号使得这种形式的代码不可能被解释成合法的声明语句。

类似地，还有其他方式通过将该语句“修饰”为非法的声明语句从而消除其二义性，不过在此我并不想讨论它们，原因很简单：那些方法只有当你和你的编译器都对标准中这个冷僻特性理解得非常清楚的情况下才管用。

准则：避开语言中那些冷僻特性，包括那些对程序员（甚至是编译器）来说模棱两可的语法结构。

要对付这种天生具有“声明—构造”二义性的语法，最彻底的方法就是完全避开它，不要使

用那些会让有些编译器误认为是声明的语法形式。打个比方，当你和别人谈话的时候，你会愿意先说一些模棱两可的话然后补充道“嗯，其实我真正的意思是……”吗？

如果能从一开始就避开这种二义性，那自然再好不过。所以我更喜欢并且推荐你使用下面的替代方案，因为它显然更容易把事情做对，而且就算是最差劲的编译器也能很好地理解这些代码的意义，此外它也更利于我们阅读代码：

```
// 示例29-3(b): 使用具名变量(推荐做法，分数: 10/10)
//
istream_iterator<string> first(cin), last;

deque<string> coll3(first, last);
```

实际上，在示例 29-3(a)和示例 29-3(b)中，我们都只需对一个参数作改动就能达到所需的效果，不过为了一致性，我们在这里对两个参数都作了处理。

准则：优先使用具名变量作为构造函数的参数。这样可以避免与声明语句产生二义性。况且这样做也能令你的代码更清晰，更易于维护。

小结

避开语言中的冷僻特性；作为程序员，我们只要知道有这些东西存在就可以了。

保持代码清晰而明确，明确地表达你的意图：在给构造函数提供参数的时候，尽可能提供具名变量作为参数，以避开语言中那些很微妙诡异的规则，同时又可以让你的代码更加干净和易于维护。

226

第 30 条　要么 double 要么彻底完蛋

难度：4

不，本条并非在讨论赌博[1]。可以说，本条要讨论的是另外一种浮点数，阅读本条可以测试你在C和C++基本浮点操作方面的技能。

JG问题

1. float 和 double 有什么区别？

Guru问题

2. 假设对于现代的桌面计算机而言，下面这段代码运行需要 1 秒，这是比较正常的：

```
int main() {
  double x = 1e8;
  while(x > 0) {
    –x;
  }
}
```

现在，如果将 double 换成 float，你觉得需要的运行时间是多少？为什么？

解决方案

float和double简述

1. float 和 double 有什么区别？

C++标准如是说：

总共有 3 种浮点类型：float、double 以及 long double。double 的精度要比 float 的高；long double 的精度要比 double 高。float 的值域是 double 值域的子集；double 的值域又是 long double 值域的子集。——[C++03, 3.9.1 ~ 8 节]

现在就让我们来看看这段定义（特别是最后一句）会对代码产生什么样的影响。 227

时间之轮

2. 假设对于现代的桌面计算机而言，下面这段代码运行需要 1 秒，这是比较正常的：

```
int main() {
  double x = 1e8;
```

1. 本条标题原文为“double or nothing”，是一个赌博术语，指等额赌博，赢了得双倍赌资，输了就输光。——译者注

```
    while(x > 0) {
      --x;
    }
  }
```

现在，如果将 double 换成 float，你觉得需要的运行时间是多少？为什么？

答案是可能 1 秒（在特定的语言实现上，float 比 double 可能更快、一样快或更慢），也可能永远，具体取决于 float 能否精确地表示从 0 到 1e8 闭区间内的所有整数。

上述摘自 C++标准的文字说明了可能存在某些可以用 double 表示而不能用 float 表示的值。特别是在一些流行的平台和编译器上，double 可以精确地表示从 0 到 1e8 闭区间内的所有整数，而 float 则不行。

如果 float 不能表示从 0 到 1e8 闭区间内的所有整数，又会如何呢？那么，float 改用的版本将从开始一直递减，最终会达到一个不能表示的值 N，其中 $N-1 == N$（由于浮点精度不够）……，于是这个循环就会永远停留在该值，直到运行这个程序的机器断电（停电或电池耗尽）或者操作系统崩溃（对于某些平台，这个可能性更大），或者直到太阳成为变星而且烧焦了内行星，甚至整个宇宙归于热寂[1]！

关于窄化转换

有些人也许会问："那么，除了宇宙热寂之外难道就没有其他问题了么？常量 1e8 的类型是 double。因此如果只是将 double 改成 float，这个程序就会因需要发生窄化转换而不能通过编译！不是吗？"对于这个问题，让我们再一次引用标准（摘自另外一节）：

228

> 一种浮点类型的右值可以转换成另一种浮点类型的右值。如果目标类型能够精确地表示源值，那么转换的结果就是那个精确表示。如果源值处于目标类型所能精确表示的两个相邻值之间，那么转换的结果具体选择哪个值，由语言实现定义（implementation-defined）。此外的行为则是未定义的。——[C++03] 4.8/1 节

这就意味着一个 double 类型的常量可以被隐式地（也就是无提示地）转换成一个 float 类型的常量，即使该转换会引起精度的损失（数据、信息、知识、状态、意义等）。之所以允许这种转换是考虑到与 C 的兼容性和可用性，但在进行浮点操作时，应紧记这些相关问题。

小结

本条仅仅是对浮点问题做了粗浅的讨论。浮点运算是一门难懂、深奥且非常晦涩的学问。可以说，世界上的人可以分为 3 种：一种知道自己不懂浮点运算（他们是对的），一种自以为懂得

1. 由于这个程序将使计算机无休止地运行，实际上是毫无必要地增加了整个宇宙的熵值，理论上加速了热寂的进程。简而言之，这样的程序对环境极其不利，应被看成是对整个人类的一种威胁！请永远不要写出这样的代码！当然，任何多余的工作都会增加宇宙的熵值，加快热寂的步伐。这也可以作为老板要求加班时你的一个不错的借口。好了，跑题的话到此为止。

浮点运算（他们是错的），还有极少的专家级人物则是想知道自己是否有可能最终完全理解浮点运算（他们是睿智的）。

准则：记住，浮点运算是神秘且深奥难懂的。在使用浮点数时请提高警惕，避免依赖于浮点转换操作。人们对算术的所有认识在面对浮点运算时几乎都或多或少存在错误。

质量不错的编译器在你试图进行一些未定义操作时会提出警告，例如，将一个比 float 所能表示的最大值还大或者比最小值还小的 double 值转换成 float；而优秀的编译器则会在你试图进行一些有定义但会损失信息精度的操作时提出警告，例如，试图将一个处于 float 所能表示的数值范围之间，却又不能被 float 精确表示的 double 值转换成 float。

对那些可能具有奇怪行为的浮点数操作要保持警惕，尽最大努力避免依赖于浮点转换操作，打开编译器的所有诊断开关，这多多少少会让你在漆黑的浮点算术世界里行进得更安全一些。 229

第 31 条　狂乱的代码

难度：4

有时候你会遇到完全不可思议的调试情形。试试下面这段代码，看看是否能找出问题的原因。

Guru问题

1. 一名程序员编写了如下代码：

```
//—文件biology.h —
//
// ……包含一些头文件，以及其他东西……
class Animal {
public:
 // 该对象的成员函数
 //
 virtual int Eat       (int) { /*...*/ }
 virtual int Excrete  (int) { /*...*/ }
 virtual int Sleep    (int) { /*...*/ }
 virtual int Wake     (int) { /*...*/ }

 // 对于那些已经有了配偶的动物，显然有时候它们会不再喜欢自己的配偶
 // 故而我们提供如下的函数:
 //
 int EatEx       (Animal* a) { /*...*/ }
 int ExcreteEx  (Animal* a) { /*...*/ }
 int SleepEx    (Animal* a) { /*...*/ }
 int WakeEx     (Animal* a) { /*...*/ }

 // ……
};

// 具体类
//
class Cat     : public Animal { /*...*/ };
class Dog     : public Animal { /*...*/ };
class Weevil: public Animal { /*...*/ };
    // ……其他动物……

// 方便但是有些多余的辅助函数
//
int Eat       (Animal* a) { return a->Eat(1); }
int Excrete (Animal* a) { return a->Excrete(1); }
int Sleep    (Animal* a) { return a->Sleep(1); }
int Wake     (Animal* a) { return a->Wake(1); }
```

然而，遗憾的是这段代码编译失败了。编译器至少拒绝了其中一个“…Ex”函数的定义，错误信息表示它已经被定义过。

为了避免这个编译错误，程序员将出错的“…Ex”函数改为注释，现在程序能够通过编译了，他开始测试“Sleep…”函数。可是，Animal::Sleep 成员函数看起来却并不总是能够正确工作。当他试图直接调用成员函数 Animal::Sleep 时，一切正常；可是当他试图通过非成员形式的 Sleep 辅助函数（该函数做的惟一事情就是调用成员形式的 Sleep）来调用成员函数 Animal::Sleep 时，有时却什么也不会发生，而且这种情况并不是每次都出现，只是偶尔出现。最后，当程序员启动调试器乃至打开链接器产生的符号映射文件试着寻找问题的根源时，他甚至发现在符号映射文件中完全找不到 Animal::Sleep 函数的任何代码！

是编译器故障吗？程序员应该给编译器开发商写一封怒气腾腾的电子邮件并同时给《纽约时报》写封信吗？这是一个滞后的千年虫问题吗？还是因为来自 Internet 上的可恶病毒？

到底发生了什么？

解决方案

1. 一名程序员编写了如下代码:

[……]

到底发生了什么？

简单地说，很多因素都会导致这种情况发生，但其中最明显最容易想到的肇事者——是的，你猜得没错——狂乱的宏和一些有意无意的重载行为所共同作用的结果！

动机

一些流行的 C++程序设计环境有时会故意提供一些宏来达到改变函数名字的目的。它们通常都有一些良好或纯洁的初衷，比如为了 API 的向前和向后兼容。例如，如果一个操作系统的某个版本中的 Sleep 函数被 SleepEx 取代，那么生产商提供的相应的头文件中很可能就会包含一些看似很有帮助的宏，自动将 Sleep 更名为 SleepEx:

231

```
#define Sleep SleepEx
```

几乎可以肯定地说，这并不是好做法。宏是封装的天敌，因为它实际影响的范围是不受控制的，即使宏的定义者也无法控制它。

宏是不负责任的

宏是一种令人讨厌、杂乱无章的东西，其原因很多，但很大一部分要归咎于这样的事实：宏是一种在预处理阶段发挥作用的文本替换设施，而在预处理阶段所有的 C++语法语义规则都还没有起作用呢。下面是宏的一些不良行为：

（1）宏会改变名字。这一行为通常是有害的。

保守一点说，这种不加提示就改变名字的行为会让调试过程显得令人迷惑不解。例如你可能认为你调用了某个函数，而实际上由于宏将名字篡改了，结果你所认为的那个函数并未被调用。

例如，考虑 Sleep 函数的非成员形式：

```
int Sleep (Animal* a) { return a->Sleep(1); }
```

在目标代码或链接器产生的符号映射文件中根本找不到 Sleep 函数的代码，因为它实际的名字是“SleepEx”。当你在挠头苦思 Sleep 函数哪里去了的时候，可能会想：“也许是编译器自动将 Sleep 函数作了内联处理”，因为这能解释为什么这个小函数在目标代码内不存在。如果得出了这个结论，并进而发送了一封怒气腾腾的电子邮件给编译器开发商，抱怨他们优化做得太过火了，那很不幸，你发脾气的对象弄错了。

有些人也许曾遇到过这个糟糕的问题。如果你也像我一样，很容易就被编译器的一些古怪行为所激怒，并且好奇心总是占上风，不满足于那些简单的解释，那么你也许会打开调试器，试图小心地进入并单步执行函数……，当执行到那个神秘函数所在的代码行（这时从源代码层面来看，该函数仍然具最初的名字），然后进入并单步执行到实际调用和完成任务的函数时，不是根据对符号映射文件的考察，这个函数却又是无论如何不存在的！）。通常这个时候你会马上意识到问题所在，并低声咒骂那些愚蠢的宏手法[1]。

232 可是，等等，情况还没那么糟糕：

（1）C++在处理名字方面有自己的一套特性。宏则提供了一条不同的途径来做类似的事情，这就导致了两者之间至少存在着有害的交互作用。

你也许会认为仅仅改变一个函数的名字并不会带来大问题。是的，你是对的；大部分情况下的确不会带来什么问题。可是如果函数的名字被改成了另一个现有函数的名字，情况又会怎样呢？C++在碰到两个具有相同名字的函数时会采取什么行动呢？当然是重载。而倘若你不能意识到这种重载是不加提示地进行，可能事情就不太妙了。

现在让我们来看看这个 Sleep 的例子。库的开发商之所以会决定提供这么一个看似很有帮助的 Sleep 宏定义，将 Sleep 自动重命名为 SleepEx，是因为实际上在库里，这两个函数早就已经存在了。考虑到这些函数也许会有不同的签名，所以我们在写自己的 Sleep 函数时，需要当心不要与库中的 Sleep 函数发生重载，要当心出现二义性或者其他问题。我们甚至偶尔会故意依赖这种重载机制来提供类似于库中的 Sleep 函数的行为。但是，如果我们的函数会不加提示地与其他函数发生重载，那么重载行为不仅不会按照所预期的那样发生，甚至即使有意重载，重载也不会发生，至少不会如我们所想像的那样发生。

对于这个问题来说，这样一个 Sleep 重命名宏可以在一定程度上解释为什么在不同的环境中

1. 因为一般来说，在调试器上一旦进入并单步执行某个函数你就会看到函数名，从而发觉原来它的名字被篡改了。——译者注

最终调用的函数不同；具体哪个函数被调用取决于在不同的调用处由特定的类型所触发的重载决议。有时调用的是我们的函数，而有时则是库。结果反正是不确定的，而且具体情况或许非常难以捉摸。

这种行为即便只发生在非成员函数身上也已经是够讨厌的了。不幸的是，就如同手榴弹爆炸后产生的弹片一样，这种危险会四散飞舞，伤及其他地方。

（2）宏不管类型

前面说过，Sleep 重命名宏的意图是改变一个全局非成员函数的函数名。然而不幸的是，这个宏会改变所有它所遇到的 Sleep 名字。如果我们凑巧有一个名为 Sleep 的全局变量，那么就连它的名字也会被不加提示地改变。一句话，这真是糟透了。

（3）宏不管作用域

更糟的是，为改变全局非成员函数名而设计的宏会将所匹配的函数（及其他）名全都改掉，不管是你自己类的成员名，还是所在名字空间内精心封装的其他名字。本例中我们编写了一个类，它包含了名为 Sleep 和 SleepEx 的成员函数；上述的许多问题在某种程度上都可以用这个 Sleep 重命名宏来解释，因为它使我们自己的函数互相之间发生了不可见的重载。实际上，因为存在前面第（1）点中提到的不可见重载问题，这也正解释了为什么有时候会调用一个意料之外的成员函数（具体调用的是哪一个，则取决于在调用处特定的类型会导致什么样的重载决议结果）。 233

你也会同意，这也是非常糟糕的。它就像一个没带手套、没有执业资格的医生（库的头文件作者），用他的脏手（宏）打开你的肢体（类或者名字空间），伸到你的身体里面将各种器官（成员以及其他的代码）重新排列……而这一切都是悄无声息地发生的（他甚至根本没意识到他正在这么干！）

小结

一句话，宏几乎就是“目空一切”，可能为所欲为。

准则： **避免使用宏。**

永远、永远、永远不要写一个名字为常见词或缩写的宏，连想都别这么想。

你对宏的本能反应应当是“宏！真恶心！”，除非在某些特定情况下有一些原因迫使你不得不使用它们，但前提是就算是用，也千万别当成一种 hack 手法来用。宏不是类型安全的，不受作用域控制……总而言之，它们彻头彻尾是不安全的家伙。如果必须写一个宏，请不要将它放在头文件中，而且应给它起一个个性化的长名字，这个名字必须足够特别，以免它的铁蹄四处践踏，“伤及无辜”。

准则： **尽量使用名字空间来封装名字。**

简而言之，请采用良好的封装形式。这不仅仅是因为封装能够产生更好的设计，更是因为它能够帮助你抵御一些未曾料到的危险性。宏是封装的天敌，因为就算是宏的定义者也无法控制宏的实际作用范围。C++中的有力工具之一就是名字空间和类，它们可以帮助管理程序中的那些应当相互无关的部分，并将它们之间的相互依赖性降至最低，适当地使用它们以及其他的一些C++设施来改善封装，不仅可以产生出更好的设计，同时也等于是提供了一把保护伞，避免你的代码受到那些考虑不周的程序员不小心丢过来的“手榴弹”的伤害。

234

第 32 条　小小的拼写错误？鬼画符似的语言以及其他奇形怪状的东西

难度：5

有时候即使是很微小的、难以察觉的拼写错误也会对代码造成巨大的影响。为了说明这种拼写错误有多难察觉，为了说明神秘的拼写错误会多么容易碰巧导致出乎意料的结果，请看下面的例子。

不用编译器，回答下列问题：

Guru问题

1. 在一个符合标准的 C++编译器上，下面程序的输出是什么？

```
// 示例32-1
//
#include <iostream>
#include <iomanip>

int main() {
 int x = 1;
 for(int i = 0; i < 100; ++i);
  // 下面这行代码会干些什么？递增??????????/
  ++x;
  std::cout << x << std::endl;
}
```

2. 在编译下面这段代码时，一个符合标准的 C++编译器会报告多少个不同的错误？

```
// 示例32-2
//
struct X {
 static bool f(int* p) {
  return p && 0[p] and not p[1:>>p[2];
 };
};
```

235

解决方案

1. 在一个符合 C++标准的编译器上，下面程序的输出是什么？

在示例 32-1 中，如果注释行最后没有不可见的空白字符，输出是 1。

代码中有两处陷阱，一处比较明显，另一处则比较隐晦。

第一，考虑 for 循环这一行：

```
for(int i = 0; i < 100; ++i);
```

这一行的结尾有个分号，这个“奇异递归拼写错误模式”[1]。会让 for 循环的循环体变成空语句。就算接下来的代码有缩进(甚至以大括号包围)，它们也不能算作 for 循环体中的内容。

这个问题是在故意转移注意力，本例中由于还有另一处陷阱，与这里的 for 循环体内是不是有任何语句其实并不重要，因为这里根本没有可以被用来重复执行的语句（虽然看起来似乎有一条）。我们来看第二点：

第二，考虑注释行。你注意到这一行以一个奇怪的“/”字符结束了吗？

```
// 下面这行代码会干些什么？递增??????????/
```

Nikolai Smirnov 写道：

> 或许这个程序中发生的事情对你来说是显而易见的，可是对我来说却不然，我在一个大程序中犯了类似的错误，结果在调试上花了数天时间。最终我发现原来我在一个注释行的结尾放了很多的问号，然而在输入最后一个问号的时候我一不小心松开了 Shift 键，结果输入了一个“/”，这样一来结尾的“??/”就被转换成了“\”，接着这个转义符就把接下来的换行符给吃掉了。——N. Smirnov，私人通信

“??/”会被转换为“\”，而后者放在一行的末尾作用相当于将接下来的一行“粘贴”到这一行的末尾！这太令人诧异了！本例中，它将接下来的“++x”粘贴到了注释行的末尾，这样一来“++x”就成了注释的一部分，从而怎么也不会被执行了。

236

有趣的是，如果翻一翻 Gnu g++文档中关于-WTRigraphs 命令行开关的描述，你会看到如下不正确的概括：

> 对注释中的三字符组[2]不会发出编译警告，因为它们不会对程序的意义产生影响[3]。

这种说法大部分时候是对的，然而我们上面给出的就是一个反例，它来自现实，不容置疑，所以说 g++文档中的说法并不总是对的。

2. 在编译下面这段代码时，一个符合标准的 C++编译器会报告多少个不同的错误？

```
// 示例32-2
//
struct X {
  static bool f(int* p) {
    return p && 0[p] and not p[1:>>p[2];
  };
};
```

1. 这里是幽默地模仿“奇异递归模板模式（CRTP）”说。——译者注
2. 三字符组（trigraph）是指三个字符组成的转义符，比如“??/”=“\”，“??!”=“~”。类似地，还有双字符组，比如“:>”=“]”。参见第 33 条。
3. 在 Google 上搜索“trigraphs within comments”，你会发现上面所说的这个描述，以及一些其他有趣的东西，在这里就不多说了。

简单的答案就是：零。这段代码完全合法并且符合标准（不管其作者是否希望它如此）。

让我们逐一来考虑那些可能会有问题的表达式，看看为什么说它们实际上都是合法的：

- 0[p]其实和 p[0]一样，都是合法的表达式。在 C（和 C++）中，形如 x[y]的表达式，只要 x 和 y 的其中一个是指针而另一个是整型值，它的意义就相当于*(x+y)。这里 0[p]和 p[0]意思相同，因为它们分别等同于*(0+p)和*(p+0)。更多的细节讨论，请参考[C99] 6.5.2.1 节。
- and 和 not 都是有效的关键字，它们分别是&&和！的替代形式。
- “:>”也是合法的。它是“]”的双字符写法，而并不是笑脸符（在 C++代码中，除注释之外的代码里，是不支持笑脸符的——真遗憾）。于是表达式的最后一部分就被解释成了“p[1]>p[2]”。 237
- 成员函数声明结尾的多余分号是完全无害的。C++类定义语法允许空的成员声明（一个单独的分号）出现在任何地方。例如下面就是一个完全合法的不含任何成员的类完全合法声明：

```
class X { ;;;;;;;;; };
```

在上面这些奇形怪状的表达式之中，大部分人似乎还是对“:>”双字符感到最为诧异。当然，很可能“:”确实是一个拼写错误，原作者实际上是想写成别的东西，例如也许是“p[1]>>p[2]”。然而最不幸的是，即便这是拼写错误，它们也是完全合法的 C++代码。 238

第 33 条 操作符，无处不在的操作符

难度：4

如果有意为之，最多可以将多少个操作符“堆”到一块？本条我们抛开产品编码的那些条条框框，去C++中找些乐子。

JG问题

1. 在一个合法的 C++程序中，如果中间不用空格隔开，最多可以将多少个加号(+)放到一起？

注意：不考虑注释、预处理指令以及宏定义中的加号，否则就太容易了。

Guru问题

2. 类似地，如果不用空格隔开，且注释中的不算，那么一个合法的 C++程序中最多能包含多少个连续的以下字符？

(a) &
(b) <
(c) |

例如，对于“&”来说，if(a&&b)这样的代码就说明一个合法的 C++程序可以包含两个连续的“&”，还能更多吗？试试看。

解决方案

背景知识：什么是贪婪匹配？它在C++编译器中起什么作用？

“贪婪匹配(max munch)”原则是说，编译器在将源代码中的字符解析成一个个的标记(token)时，会使用一种贪婪匹配算法，也就是说会尽可能让一个标记包含更多的字符。因此“>>”会被解析成一个标记，从而其意义也就成了流提取操作符，而不会被解析成两个“>”，即便是对于下面这段代码也是如此：

```
template<class T = X<Y>> ... // 最后的“>>”被解析为单个token，编译错误!
```

这也正是为什么在写上面的代码时必须如下增加一个空格的原因：

239

```
template<class T = X<Y> > ...
```

类似地，“>>>”会被解释成一个“>>”后跟一个“>”，而不是一个“>”后跟一个“>>”。

操作符的乐趣

1. 在一个合法的 C++程序中，如果中间不用空格隔开，最多可以将多少个加号(+)放到一起？

注意：不考虑注释、预处理指令以及宏定义中的加号，否则就太容易了。

我们可以建立一个源文件，其中包含任意长度的“+”序列，直到达到编译器所能处理的极限为止（例如达到了编译器所能处理的源文件的最大行长度）。

如果序列中包含偶数个“+”号，那它就会被解释成“++ ++ ++ ++ ... ++”，即由多个“++”组成的序列。要让这个序列正常工作并具有合法的语义，只需定义一个可**级联**的、具有用户自定义前缀++操作符的类即可，例如：

```
// 示例33-1(a)
//
class A {
public:
  A& operator++() { return *this; }
};
```

现在就可以这样写（如果真的想这么做）：

```
A a;
++++++a;        // 含义为: ++ ++ ++ a;
```

它的意思是：

```
a.operator++().operator++().operator++()
```

那么，如果序列中包含奇数个“+”号，情况又会怎么样呢？答案是序列将会被解释成“++ ++ ++ ++ ... ++ +”，即一系列“++”后跟着一个“+”作为结尾。要使其能正常工作，只需为我们的类添加一个 operator+()重载。例如：

```
// 示例33-1(b)
//
class A {
public:
  A& operator+ () { return *this; }
  A& operator++() { return *this; }
};
```

现在就可以写如下形式的代码了（如果你觉得这个下午特别漫长，需要排遣一下的话）：　240

```
A a;
+++++++a;        // 含义为: ++ ++ ++ + a;
```

它的意思是

```
a.operator+().operator++().operator++().operator++()
```

这个技巧非常简单。对于其他操作符来说要创建超长的序列似乎有点挑战性，不过仍然是可能的。

操作符的误用

示例 33-1(a)和示例 33-1(b)中的代码并没有特别地误用++和+操作符的常规语义。然而我们接下来要做的事情，是你在产品代码中无论如何也不希望看到的，这里纯粹是为了好玩而已。

2. 类似地，如果不用空格隔开、且注释中的不算，那么一个合法的 C++程序中最多能包含多少个连续的以下字符？

（a）&;　　（b）<;　　（c）|

对于这个问题，我们创建一个如下所示的辅助类：

```
// 示例33-2
//
class A {
public:
  void operator&&(int) { }
  void operator<<(int) { }
  void operator||(int) { }
};

typedef void (A::*F)(int);
```

现在就让我们来迎接挑战吧：

（a）&

答案：5 个。

嗯，很显然“&&”是合法的，而让“&&&”合法也没多大困难，因此让我们进入下一层：我们可以建立一个连续 4 个&字符组成的序列吗（即“&&&&”）？如果可以，它将会被解释为“&&&&”。然而，形如“a && && b”的表达式在语法上却又是不合法的，因此我们不能让两个二元操作符“&&”紧连在一起。

241

关键是要看到，可以将第二个“&&”看作是操作符调用，而让前面的“&&”成为操作符调用之外的什么东西的“结尾”。考虑到这一点我们就不难看出，第一个&&可以作为一个名字的结尾，明确地说就是一个函数名的结尾。因此，我们所需要的就是这样一个&&操作符：它能够接受一个指向其他&&操作符的指针作为其第一个参数：

```
void operator&&(F, A) { }
```

这么一来就可以写：

```
&A::operator&&&&a;        // && &&
```

它的意思是：

```
operator&&(&A::operator&&, a);
```

这已经是我们所能写出的偶数个&的最长序列了，因为“&&&&&&”肯定是不合法的。为什么？因为它将被解释为“&& && &&”，就算我们可以再次将第一个&&作为某个函数名的一部分，我们也无法将最后的那个&&作为某个名字的开始，因此最终必然会导致两个二元&&操作符连在一起，而这是非法的。

可是我们还能再多加入一个&符号，得到奇数个&的序列吗？是的，可以。显然，“&&&&&”

的意思是“&& && &”；我们已经为前一个问题给出了答案，所以稍加思考就会发现添加一个&其实很容易：

```
&A::operator&&&&&a;      // && && &
```

我们使用了进行取址操作的内建&操作符，所以上面的代码就相当于：

```
operator&&(&A::operator&&, &a);
```

现在来看看（b）和（c）：

（b）<

（c）|

这两个问题的答案都是：4 个。

既然已经知道了（a）的做法，那么这次应该轻车熟路了。要想得到 4 个连续的“<”或“|”序列，只需重复前面采用过的手法，如下：

```
void operator<<(F, A) { }
void operator||(F, A) { }
```

这么一来就可以写：

```
&A::operator<<<<a;      // 意为“<< <<”
&A::operator||||a;      // 意为“|| ||”
```

它们的意思分别是：

242

```
operator<<(&A::operator<<, a);
operator||(&A::operator||, a);
```

以上已经是我们所能得到的偶数个符号的最长序列了，因为“<<<<<<”和“||||||”都是不合法的，道理和“&&&&&&”一样。只不过不同的是，这一次无法再加入一个额外的“<”或“|”来得到一个五字符的序列，因为并不存在一元的“<”或“|”操作符。

附加题

下面是一个附加题：在一个合法的 C++程序中，一行中最多可以有多少个连续的问号（?）？

继续阅读之前请先思考一下。它比看上去要难得多。

——

有答案了吗？

也许你会认为答案是“1”，因为 C++语言中问号惟一合法的用途就是在三元操作符“?:”中。的确，在所有合法的语言特性中只有这个操作符包含了一个问号。然而别忘了，在翻译器和预处理器中还存在着一些在语言语法规则里怎么也想不到的东西。C++中尤其如此，除了语言之外，甚至预处理器中也有很多。

对于问号来说，正确的答案应当是3，例如：

```
1???-0:0;
```

这个问题比本条的其他所有问题都难，因为它是惟一一个不遵守“贪婪匹配”原则的。3个连续的问号并非被解析成“??”和“?”。为什么呢？因为“??-”凑巧是一个三字符组，而后者在源文件处理的早期就被替换掉了（在编译器进行标记化之前，甚至是在处理预处理指令之前）！如果你还未曾听说过三字符组，别担心，这只不过说明你还未曾使用过一些非常特殊的外国键盘，或者你还没有阅读第32条。三字符组的出现是为了为某些非常规的源代码字符提供替代写法（它们是：#、\、^、[、]、|、{、}和~）。如果你的键盘上很不巧没有这些字符键，三字符组就有用武之地了。

因此，对于本例来说，远在编译器进行任何标记化工作之前，三字符组“??-”就被替换成了“~”，因此上面那条语句就变成了：

```
1?~0:0;
```

243 它被标记化为：

```
1 ? ~ 0 : 0 ;
```

其含义为：

```
1 ? (~0) : 0 ;
```

小结

三字符组是从C语言继承过来的一个特性，它们在现实当中很少被用到。只要举一个例子你就会明白它们有多么不常用了：到写作本书时，我所知道的一些编译器默认都不打开三字符组支持，其中就有一个编译器的文档这样来描述这个选项：“支持不合需要的、极少使用的ANSI三字符组特性”。这一说法真是中肯之极！

244

风格案例研究

剖析已经发布的代码可能显得有点无礼，确实有点，不过很有意思。

该部分引入了一个新的主题：我们将会考察一些现实世界中的、已经公开发布的代码，评价它们，通过对代码中适当与欠缺的方面的评析来阐释恰当的设计和编码风格，并利用在这个过程中得到的认知来对代码进行改造，最终得到一个改进后的版本。看完之后你或许会惊奇于那些由专家之手编写、测试并审查过的代码居然也可以做这么多的改进。

不过，在享受阅读乐趣的同时，别忘了想想在自己的源代码控制系统中已经签入的那些代码中存在多少这类问题？

第34条 索 引 表

难度：5

索引表确实是一种有用的惯用法，而且是一种值得去了解学习的技术。但如何才能有效地实现这一技术呢……应当不仅是“有效”，“完美”怎么样？

JG问题

1. 谁会受益于清晰易懂的代码？

Guru问题

2. 以下代码展示了在已有容器中创建索引表的一种有趣且有用的惯用法。如需更详细的解释，请参考其原文[Hicks00]。

评价这段代码并找出：

（a）像无效语法或不可移植的风格习惯之类的“机械”错误。

（b）在风格上可以做哪些改进，使代码的清晰度、重用性和可维护性都得到改善。

```
// sort_idxtbl(...)的作用是排列一个索引数组
#include <vector>
#include <algorith>
template <class RAIter>
struct sort_idxtbl_pair
{
  RAIter it;
  int i;
  bool operator<( const sort_idxtbl_pair& s )
    { return (*it) < (*(s.it)); }
  void set( const RAIter& _it, int _i ) { it=_it; i=_i; }
  sort_idxtbl_pair() {}
};
template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
{
  int iDst = last-first;
  typedef std::vector< sort_idxtbl_pair<RAIter> > V;
  V v( iDst );
  int i=0;
  RAIter it = first;
  V::iterator vit = v.begin();
  for( i=0; it<last; it++, vit++, i++ )
```

246

```
    (*vit).set(it,i);
  std::sort(v.begin(), v.end());
  int *pi = pidxtbl;
  vit = v.begin();
  for( ; vit<v.end(); pi++, vit++ )
    *pi = (*vit).i;
}
main()
{
  int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };
  cout << "#################" << endl;
  std::vector<int> vecai(ai, ai+10);
  int aidxtbl[10];
  sort_idxtbl(vecai.begin(), vecai.end(), aidxtbl);
  for (int i=0; i<10; i++)
  cout << "i=" << i
       << ", aidxtbl[i]=" << aidxtbl[i]
       << ", ai[aidxtbl[i]]=" << ai[aidxtbl[i]]
       << endl;
  cout << "#################" << endl;
}
```

解决方案

清晰度

1. 谁会受益于清晰易懂的代码？

简而言之，对所有人都有好处。

首先，清晰的代码更易于调试，也正因为清晰，所以代码在第一时间的错误也就少很多，就眼前来说，编写清晰的代码也至少可以让你的生活更轻松一些（相关案例，请参见第 27 条围绕示例 27-2 的讨论）。此外，当一个月或一年之后重读你的代码时（如果代码当初没问题并投入了实际使用，这一环节通常是免不了的），你就会发现更容易“重拾”那些清晰的代码，明白代码都干了些什么。绝大多数程序员觉得要在头脑中记住代码的全部细节并保持哪怕只是几周时间都是很困难的，尤其是在转向其他的工作之后，要记住这些就更加困难了。经过几个月乃至几年之后，即便是重读自己以前写的代码，也很容易觉得那似乎是一个陌生人写的（只不过那个“陌生人”恰好与你有同样的个人编码风格）。 247

利己方面已经说得够多了。让我们来看看有利于别人的方面：代码维护者也将获益于代码的清晰性和可读性。毕竟，要将代码维护好首先得“投入”（grok）代码。Robert Heinlein 杜撰了“grok”一词，意指深入而完整地理解；在此处，这个词还包含有理解代码本身内在的工作方式、代码的副作用以及代码与其他子系统的交互方式的意思。总而言之，在没有完全理解一段代码的情况下

就贸然去修改它，太容易引入新的错误。清晰、可理解的代码更容易让人投入其中，因此，对于这种代码的修补就变得不那么脆弱、危险，而且也不太容易无意间引入不想引入的副作用。

然而，最重要的一点是，由于以上这些原因，你的最终用户将得益于清晰、可理解的代码：这种代码从一开始错误就很少；更容易被正确地维护，而且在维护过程中也不至于引入同样多的错误。

准则：一般来说，优先考虑编写清晰、正确的代码。

深入剖析索引表

2. 以下代码展示了在已有容器中创建索引表的一种有趣且有用的惯用法。如需更详细的解释，请参考其原文[Hicks00]。

评价这段代码并找出：

（a）像无效语法或不可移植的风格习惯之类的“机械”错误。

（b）在风格上可以作哪些改进，使代码的清晰度、重用性和可维护性都得到改善。

请允许我重复一遍：这些代码展示了一种有趣且有用的惯用法。我常常发现必须以不同的方式访问相同的容器，例如按照不同的排序准则来看待同一个容器中的元素。所以说这样的方法的确是很有用的：以一个主容器（譬如 vector<Employee>）保存实际数据，以若干副容器保存指向主容器中元素的迭代器，这样一来我们就能支持各式各样的访问方式（例如，set<vector<Employee>::iterator, Funct>，其中 Funct 是用以间接比较 Employee 对象的仿函数，于是就可以产生不同于对象在 vector 中物理存储顺序的排序）。

248

话虽如此，风格也是很重要的。原作者爽快地允许我将他的代码作为相关案例，而我也并非在此对他的代码妄加评论，只不过是通过剖析和评论已发布代码的方式来阐释编码风格原则罢了，很早以前诸如 P.J.Plauger 等人就已经这么做了。在此之前我评论过其他人发布的东西，同时也让别人来批评我的东西，而且我相信这一做法会得到更多人的认同和效仿。

说完了所有这些，让我们来看看究竟能对给出的代码案例作哪些改进。

更正“机械”错误

（a）像无效语法或不可移植的风格习惯之类的“机械”错误。

建设性批评的首要方面就是代码中的“机械”错误。像下面列出的这些“机械”错误在大多数平台上是无法通过编译的。

```
#include <algorith>
```

（1）正确拼写标准库头文件名。在这个例子中，头文件<algorithm>被误拼作<algorith>。我首先猜测，这也许是因为原先代码的测试环境是一个 8 字符文件名的系统，但即便是我的老版本

Windows（基于 8.3 文件名系统）上的老版本 VC++也无法编译这样的代码。就算是在那些限制颇多的文件系统上，编译器本身也是被要求能支持任何标准长度的头文件名的，即便编译器只不过是在背后悄悄将它映射成较短的文件名（或根本不映射到文件上）。

接下来，考虑：

```
main()
```

（2）*正确地定义 main 函数*。这种无修饰的 main 函数原型从来都不是标准 C++[C++98]风格的，虽说它也可以作为一个合法的编译器扩展特性（前提是编译器得给出警告）。这种 main 函数原型在 C99 之前是有效的，因为当时的 C 里面允许所谓的“隐式 int 声明”，但这在 C++（C++里从来就不允许隐式 int）和 C99[C99]（C99 甚至果断地将这一特性彻底从标准中剔除掉了）中都是非标准的。在 C++标准中，请参见：

- 3.6.1/2 节：可移植代码必须将 main 定义为 int main()或 int main(int,char*[])两种形式中的一种。
- 7/7 节脚注 78 和 7.1.5/2 节脚注 80：隐式 int 是被禁止的。
- 附录 C（兼容性），对 7.1.5/4 的注释：明确指出 main()这种形式在 C++中是无效的，必须写作 int main()。

249

准则：不要依赖隐式int；这不是符合标准的可移植的C++。特别地，“void main()”或“main()”从来就不是标准C++写法，虽然仍有很多编译器将它们作为扩展加以支持。

```
cout << "##################" << endl;
```

（3）*永远记得 # include 所需要的类型定义的头文件*。这个程序使用了 cout 和 endl 但却没有#include<iostream>。那为什么这在代码原作者的系统上能工作呢？这是因为 C++标准头文件会互相#include，但不像 C，C++并没有指定哪些标准头文件#include 哪些其他标准头文件。

在这个案例中，程序有#include <vector>和<algorithm>，而在原来的那个系统上，也许恰好某个头文件间接地#include <iostream>了。这在原代码所使用的特定的库实现上或许是行得通的，甚至在我这恰巧也能正常工作，但它并不是可移植的，而且也不是一种好风格。

（4）*遵循 More Exceptional C++ [Sutter02]的 39 条中关于使用名字空间的原则*。就 cout 和 endl 而言，程序必须以 std::限定它们，或者写作：using std::cout; using std::endl;。不幸的是，忘记名字空间域限定符的情况仍然很普遍。我得赶紧指出，这段代码的原作者对 vector 和 sort 做了正确的名字限定，这倒很不错。

改进风格

（b）*在风格上可以做哪些改进，使代码的清晰度、重用性和可维护性都得到改善*。

除了“机械”错误之外，在代码案例中还有些地方，如果让我来实现，我会以不同的方式来完成。首先，要对那个辅助性结构 sort_idxtbl_pair 做两点评价：

```
template <class RAIter>
struct sort_idxtbl_pair
{
  RAIter it;
  int i;
  bool operator<( const sort_idxtbl_pair& s )
   { return (*it) < (*(s.it)); }
  void set( const RAIter& _it, int _i ) { it=_it; i=_i; }
  sort_idxtbl_pair() {}
};
```

250

（1）务必保证 const 正确性。在此例中，sort_idxtbl_pair::operator< 并不会修改*this，所以它应当被声明为 const 成员函数。

准则： 确保const正确性。

（2）消除冗余代码。该程序显式地写出了类 sort_idxtbl_pair 的默认构造函数，然而它却与隐式生成的版本没有丝毫区别。所以可以省去。此外，sort_idxtbl_pair 的两个数据成员本就是公用的，那个 set()成员函数虽说可以让设置这两个成员时的语法稍微好看一点，但由于它只在一处调用过，因此就为这点好处并不值得引入这一额外的复杂性。

准则： 避免代码重复和冗余。

下面我们进入核心函数 sort_idxtbl()：

```
template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
{
    int iDst = last-first;
    typedef std::vector< sort_idxtbl_pair<RAIter> > V;
    V v( iDst );
    int i=0;
    RAIter it = first;
    V::iterator vit = v.begin();
    for( i=0; it<last; it++, vit++, i++ )
        (*vit).set(it,i);
    std::sort(v.begin(), v.end());
    int *pi = pidxtbl;
    vit = v.begin();
    for( ; vit<v.end(); pi++, vit++ )
        *pi = (*vit).i;
}
```

（3）选择有意义且合适的名字。在这个案例中，sort_idxtbl 是个具有误导性的名字，因为这

个函数并不是在对一个索引表进行排序，而是创建了一个索引表！另一方面，代码使用了模板参数名字 RAIter 来指出这是个随机访问的迭代器，这一点可以得个不错的分数；因为这个随机访问特性正是该版本的代码所必需的，所以将模板参数命名为 RAIter 可以作为一个很好的提示。

准则：选择清晰而有意义的名字。

251

（4）确保一致性。在 sort_idxtbl()中，有时变量是在 for 循环初始化语句中被置初值的，而有时却又不是这样。这就使得代码更难以阅读，至少对我来说是这样。这一点所带来的好处因人而异。

（5）消除不必要的复杂性。这个函数里面有好几处使用了不必要的局部变量！第一，变量 iDst 被初始化为 last-first，但在后面却只被使用了一次；为什么不在使用处直接写 last-first 来摆脱这种混乱的状况呢？第二，vector 的迭代器 vit 被创建的地方其实完全可以使用下标索引来代替，效果一样，而且代码也会更清晰些。第三，局部变量 it 被初始化为函数参数 first 的值，而在此之后 first 本身就再没被使用过；我个人倾向于直接使用函数参数（即使改变参数的值也完全没有问题，因为是按值传递的），而不是引入另一个名字。

（6）复用（第一部分）：更多地复用标准库。现在，原先的程序因为复用 std::sort 而获得高分，这挺不错。但是为什么不采用 std::copy 而要手工实现最后的循环来完成复制工作呢？为什么要重新做一个只比 std::pair 多一个比较函数的专用的 sort_idxtbl_pair 呢？除了写起来更简单，复用还使代码更具可读性。谦虚一点，复用已有的代码吧！

准则：了解并在任何适当的地方使用（复用）标准库设施，而不是自己去手动实现。

（7）复用（第二部分）：令实现本身更易于被复用（一石二鸟）。原先的代码中除了函数本身之外并没有任何东西是可以直接可复用的。辅助类 sort_idxtbl_pair 与它的用途捆绑得太紧，不是独立可复用的。

（8）复用（第三部分）：改进函数的原型。原先的函数原型如下：

```
template<class RAIter>
void sort_idxtbl(RAIter first, RAIter last, int* pidxtbl)
```

它用一个光秃秃的 int*指针指向输出区，而我通常会避免这样做，我更倾向于使用托管的存储空间（比如 vector）。不过有一点是明确的：最终用户要能够调用 sort_idxtbl 并将输出放到一个普通数组或一个 vector 或是其他什么东西里。很明显，“能够将结果输出到任何容器”的要求正是迭代器的用武之地，不是吗？（参见第 5 条和第 6 条）。

```
template< class RAIn, class Out >
void sort_idxtbl( RAIn first, RAIn last, Out result )
```

252

准则：避免不必要的类型硬编码，从而扩展泛型组件的可复用性。

（9）复用（第四部分），或者叫“尽量使用!=来比较迭代器”：当进行迭代器的比较时，务必使用!=（!=对各种类型的迭代器都可用）而不是<（<只对随机访问迭代器有效），当然除非你真的一定要使用<而且刻意只支持随机访问迭代器。原程序中使用了<来比较迭代器，这对于随机访问迭代器没有问题，而程序的本意也就是创建索引表放入 vector 和数组中，而这两者都支持随机访问迭代。但是，我们没理由不想让这些功能可以作用在诸如 list 和 set 的其他不支持随机访问迭代的容器上啊！而原代码之所以不能适用于这些容器正是因为使用了<而不是!=来比较迭代器所致。

正如 Scott Meyers 在[Meyers96]第 32 条中阐述的，“在‘将来时态’下编程。”他论述道：

好的软件能够适应变化。它能够容纳新的特性，移植到新的平台，满足新的需求，处理新的输入。具有这等灵活性、健壮性、可靠性的软件并非轻易得来的。它是程序员们在满足了现在的需求并关注了将来的可能性之后设计和实现出来的。这样的软件能够温和地适应变化，是那些在未来时态下开发程序的人写出来的。

准则：尽量使用 != 而非 < 来比较迭代器。

（10）除非真的需要旧的值，否则尽量使用前置递增。这里，对于迭代器而言，应当习惯性地使用前置递增（++i），避免后置递增（i++）；参见[Sutter00]第 6 条。诚然，对于原先的代码来说这两种做法或许不会带来本质上的差别，因为 vector<T>::iterator 很可能是复制开销很小的 T*（尽管也许不一定是这样——这是由 STL 实现决定的），但如果我们要落实第（9）条建议，就不能将眼光局限在 vector<T>::iterator 上面，况且大多数其他的迭代器都是类类型的，或许其复制的开销仍然不算很昂贵，可我们何必要无谓地引入这一潜在的效率损失呢？

准则：尽量选择使用前置递增而非后置递增，除非确实需要使用旧的值。

253　上面几点已经覆盖了大多数重要的问题。不过还是有一些可以拿来评论的东西，只是我觉得没有必要把注意力放在这些不是很重要的问题上。例如，产品代码中应当有说明类和函数的用途及语义的注释，但这不适用于杂志文章所附的代码，因为文章里会有语言组织得更好、更为详细的解释。

我故意没有对主函数的风格进行评价（这跟上面提到的“机械”错误有所区别，后者如果不加以更正的话将会导致代码无法通过编译），因为毕竟这个主函数只是个演示的工具，它帮助读者了解索引表这个设施是如何使用的，而索引表本身才是焦点所在。

小结

让我们在保持原代码的接口的基础上换一种设计[1]。把我们的评论限定在改正代码中的“机

---

1. 原作者（[Hicks00]的作者）也报告了来自另一个读者的反馈，他展示了另一种优雅的，但完全不同的方法：他创建了一个类似容器的对象，该对象包含了原始容器及其迭代器，而且允许使用不同顺序的迭代访问（实际上，Boost 库（www.boost.org）中已经有了一个非常重量级的实现，称为 multi_index_container——译者注）。

械”错误和基本风格的范围内，然后，考虑下面三种改进版本。它们每一个都有其各自的优点、缺点以及风格偏好，在代码注释中有相应的解释。这三个版本的共同点就是更为清晰、更易理解和更适于移植，这三点对你我的公司来说应该是很有价值的。

```
// [Hicks00]中的代码的一个改进版本
//
#include <vector>
#include <map>
#include <algorithm>
// 解决方案1: 进行了一些基本的清理工作，但仍保留了原先代码的大致结构。
// 代码量缩减至17行（即使你把public:和private:也算上），而原先的代码有23行。
//
namespace Solution1 {
 template<class Iter>
 class sort_idxtbl_pair {
 public:
   void set( const Iter& it, int i ) { it_ = it; i_ = i; }
   bool operator<( const sort_idxtbl_pair& other ) const
     { return *it_ < *other.it_; }
   operator int() const { return i_; }
 private:
   Iter it_;
   int i_;
 };
 // 绝大多数改动都在该函数中，你可以看到，该函数现在只有5行，而原先有13行。
 // 在每行代码之后，我给出了原来的代码以供比较。
 // 尽量编写清晰而简洁的代码，不要引入不必要的复杂性和模糊性!
 //
 template<class IterIn, class IterOut>
 void sort_idxtbl( IterIn first, IterIn last, IterOut out ) {
   std::vector<sort_idxtbl_pair<IterIn> > v(last-first);
     // int iDst = last-first;
     // typedef std::vector< sort_idxtbl_pair<RAIter> > V;
     // V v(iDst);

   for( int i=0; i < last-first; ++i )
     v[i].set( first+i, i );
     // int i=0;
     // RAIter it = first;
     // V::iterator vit = v.begin();
     // for (i=0; it<last; it++, vit++, i++)
     // (*vit).set(it,i);

   std::sort( v.begin(), v.end() );
     // std::sort(v.begin(), v.end());
```

254

```
    std::copy( v.begin(), v.end(), out );
      // int *pi = pidxtbl;
      // vit = v.begin();
      // for (; vit<v.end(); pi++, vit++)
      // *pi = (*vit).i;
  }
}
// 解决方案2: 使用std::pair而不是重新发明一个类似pair的辅助类。
// 现在代码从原先的23行降到了12行。其中有8行的意图是特定于本问题的，剩下4行是可复用于
// 其他情况下的。
//
namespace Solution2 {
  template<class T, class U>
  struct ComparePair1stDeref {
    bool operator()( const std::pair<T,U>& a, const std::pair<T,U>& b ) const
      { return *a.first < *b.first; }
  };
  template<class IterIn, class IterOut>
  void sort_idxtbl( IterIn first, IterIn last, IterOut out ) {
    std::vector< std::pair<IterIn,int> > s( last-first );
    for( int i=0; i < s.size(); ++i )
      s[i] = std::make_pair( first+i, i );

    std::sort( s.begin(), s.end(), ComparePair1stDeref<IterIn,int>() );

    for( int i=0; i < s.size(); ++i, ++out )
    *out = s[i].second;
  }
}

// 解决方案3: 只是为了展示一些细节方面的替代做法，下面的代码中使用了一个multimap来消除
// 单独的排序步骤，并使用std::transform()来代替手写循环。仍然有13行代码，不过更多代码成为
// 了可复用的。该方案使用了更多空间开销，而且很可能时间开销也更大，因此我倾向于使用解决
// 方案2。只不过这个方案展示了寻找问题的替代方案的过程。
//
namespace Solution3 {
  template<class T>
  struct CompareDeref {
    bool operator()( const T& a, const T& b ) const
      { return *a < *b; }
  };

  template<class T, class U>
  struct Pair2nd {
    const U& operator()( const std::pair<T,U>& a ) const { return a.second; }
  };
```

255

```
  template<class IterIn, class IterOut>
  void sort_idxtbl( IterIn first, IterIn last, IterOut out ) {
    std::multimap<IterIn, int, CompareDeref<IterIn> > v;
    for( int i=0; first != last; ++i, ++first )
      v.insert( std::make_pair( first, i ) );
    std::transform( v.begin(), v.end(), out, Pair2nd<IterIn const,int>() );
  }
}
// 测试用例基本没有变动，只不过输出结果被导入到输出迭代器（原来是一个int*）所指的区间（这
// 里是一个vector）中去，同时直接利用ai作为容器。
//
#include <iostream>

int main() {
  int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };

  std::cout << "##################" << std::endl;
  std::vector<int> aidxtbl( 10 );

  // 使用另一个名字空间的名字以测试不同的解决方案。
  Solution3::sort_idxtbl( ai, ai+10, aidxtbl.begin() );
  for( int i=0; i<10; ++i )
  std::cout << "i=" << i
            << ", aidxtbl[i]=" << aidxtbl[i]
            << ", ai[aidxtbl[i]]=" << ai[aidxtbl[i]]
            << std::endl;
  std::cout << "##################" << std::endl;
}
```

256

257

第35条 泛型回调

难度：5

泛型代码的吸引力部分来源于它的可用性以及在尽可能多的合理情况下的复用性。通过本条将会了解如何对下文引用到的简易回调类进行一些风格上的改进，以及如何使其更有用并真正成为泛型且广泛适用的代码。

JG问题

1. 在设计和编写泛型设施的时候有哪些品质方面的要求？请解释。

Guru问题

2. 下面的代码展示了一个用于外覆回调函数的惯用手法，它不但有趣而且真正有用。要想知道更为详细的解释，可以参考作者的原文[Kalev01]。

评价这段代码，指出：

（a）在风格方面可以作哪些改进来更好地贴合习惯的C++用法。

（b）是否有某些“机械性的”限制影响了该设施适用的广泛型性。

```
template < class T, void (T::*F)() >
class callback
{
public:
  callback(T& t) : object(t) {}          // 将实际的对象赋给T
  void execute() {(object.*F)();}        // 调用回调函数

private:
  T& object;
};
```

258

解决方案

泛型的品质

1. 在设计和编写泛型设施的时候有哪些品质方面的要求？请解释。

泛型代码首先应当是可用的。但这并不意味着它必须万求万应，把什么情况都给考虑进去，而是意味着泛型代码的编写者应当作一些合理且各方面兼顾的考虑，以避免如下三件事情：

（1）避免不恰当的类型限制。例如，假设你编写的是个泛型容器，那么要求容器中的对象具有复制构造函数和不抛异常的析构函数是完全合理的。但默认构造函数和赋值操作符呢？许多有

用的类型并没有默认构造函数，而人们又想将它们放入容器中，因此如果我们的容器加上这条限制，就等于将这类（没有默认构造函数的）类型排除掉了。这可不太“泛型”（参见 *Exceptional C++* [Sutter00]第 11 条及其上下文中的一个完整的例子）。

（2）*避免不合理的功能限制*。如果你写的是一个能做 X 和 Y 这两件事的设施，那如果有些用户想要做 Z 事，而且 Z 跟 Y 并没太大差别，这时候怎么办？有时你会想让你的设施变得足够灵活以便支持 Z，有时你不会。好的泛型设计的一部分是选择便于设施被定制或扩展的方法和方式。这在泛型设计中重要的一点应当没什么可意外的，因为同样的原则对于面向对象的类设计来说同样有效。

基于策略的设计是几个重要的技术之一，用它可以设计出具有“可插拔”行为的泛型代码。具体例子可参考[Alexandrescu01]中的部分章节；从 SmartPtr 和 Singleton 这两个章节开始是个不错的选择。

这就带来了一个相关的问题：

（3）*避免不合理的单片式设计*。这一重要问题在我们下面要考虑的风格案例中并没有直接体现出来，况且也值得花一些篇幅来就其本身进行讨论，因此在后面我为它设置了专门的条，而且还是一个系列：第 37 条至第 40 条。

在以上三点中，你会注意到一个不断重复的词：不合理的。顾名思义，在足够泛型（避免类似“我确信没人会用除 char 之外的任何类型来实例化它”这样的预见）与过度工程（避免“如果有人想要用这个烤箱 LED 显示器驱动程序去控制太空船的助推器闭火怎么办？”这类妄想症）之间找到合适的点需要良好的判断力。 259

剖析泛型回调

2. *下面的代码展示了一个用于外覆回调函数的惯用手法，它不但有趣而且真正有用。要想知道更为详细的解释，可参考作者的原文*[Kalev01]。

下面重复一下前面的代码：

```
template < class T, void (T::*F)() >
class callback
{
public:
  callback(T& t) : object(t) {}          // 将实际的对象赋给T
  void execute() { (object.*F)(); }      // 调用回调函数

private:
  T& object;
};
```

说真的，在一个仅有两个只占单行的成员函数的简单类中能有多少错误呢？呃……正如它所表现出来的，它的极度简单性正是问题之一。该类模板不必是重量级的，完全不必，但它至少不应该像这么简陋。

风格改进

评价这段代码，指出：

（a）在风格方面可以作哪些改进来更好地贴合习惯的 C++用法。

你能够找出哪些？下面就是我所能找出来的地方：

（1）构造函数应当为 explicit 的。作者或许并不想提供从 T 到 callback<T>的隐式转换。良好设计的类应当避免给它们的用户带来这类潜在问题的可能。因此我们真正想要的更像是这样：

```
explicit callback(T& t) : object(t) {} // 将实际的对象赋给T
```

既然我们已经看到了这一行，那我就不妨指出其中的另一处风格问题，它本质上非关设计，而是关于描述：

准则：尽量使用explicit构造函数，除非真的想使该构造函数能够用于类型转换。

（2）（瑕疵）注释有问题。其中的“赋给”一词不恰当，某种程度上带有误导性。更恰当的说法是：在构造函数中，我们是将一个 T 对象“绑定到”引用 t。此外，虽说读了好几遍，但我还是不明白“赋给 T”这话是什么意思。因此更好的注释仍然应当是“绑定实际的对象”。

260

```
explicit callback(T& t) : object(t) {} // 绑定实际的对象
```

不过话说回来，就连这句注释也是废话，因为代码中已经说得明明白白了，这就让这句多余的注释听起来有点可笑且毫无用处，因此最好的做法是干脆把它删掉：

```
explicit callback(T& t) : object(t) {}
```

（3）execute()成员函数应当为 const 的。毕竟 execute()函数并没有对 callback<T>对象的状态进行任何改变。这又是个基本问题：const 正确性或许是老生常谈，但它的确是个好习惯。至少从 20 世纪 80 年代早期开始 const 正确性的价值就已经在 C/C++中确立了，而 2000 年以后随着我们开始编写大量的模板代码，const 正确性的价值丝毫没有消失。

```
void execute() const { (object.*F)(); }          // 调用回调函数
```

准则：确保const正确性。

既然我们已经批驳了 execute()函数，那么不妨指出其中的一些可以说是更为严重的习惯问题：

（4）（惯用法）execute()函数应该写成 operator()。在 C++中，人们习惯于使用函数调用操作符（operator()）来执行一个函数调用式的操作。实际上 execute()后面的注释本来就显得有点多余，而这么一来就完全没必要了，因此我们也干脆将它去掉，因为现在我们的代码已经符合了人们的习惯，能够完全描述它自己了。下面的代码说明了这一点：

```
void operator()() const { (object.*F)(); }        // 调用回调函数
```

“但……”你可能会问：“如果我们提供了函数调用操作符，这个类不就成了函数对象了吗？”完全正确！这就令我们忍不住想，既然 callback<T>是函数对象，那么或许它也应该是可适配的。

准则：为惯用的函数对象提供operator()，而不是具名的execute()函数。

（5）缺陷：（惯用法）这个回调类应当继承自 std::unary_function 吗？[Meyers01]的第 36 条对可适配性进行了更为详细的讨论，并说明了可适配性为什么通常是好事。然而可惜的是，这里的回调类却不应该继承自 std::unary_function，至少目前还不能，对此我们有两点有力的理由： 261

- 首先，它并非一元函数。它不接受任何参数，而一元函数是指接受惟一一个参数的函数（不，void 不算）。
- 而且不管怎样从 std::unary_function 继承都算不上可扩展的做法。后面我们会看到，回调或许应当能够接受其他类型的函数，取决于其参数个数的不同，某些情况下或许回调根本就无法继承某个来自标准里的基类。例如，倘若让回调类为支持三个参数的函数，那么就没有所谓的 std::ternary 可以作为基类了。

从 std::unary_function 或 std::binary_function 进行继承可以为回调类方便地添上一些重要的 typedef，约束器以及其他类似的设施需要这些 typedef 才能工作，但也只有当需要将回调跟这类设施一起使用时，继承才有意义。不过由于这种回调类的特质以及它们被使用的方式，产生这一需求的可能性微乎其微（如果以后人们发现它的单参或双参版本应当可以跟约束器一起使用，那么只要让单参和双参版本分别继承自 std::unary_function 和 std::binary_function 即可）。

纠正“机械性”错误及限制

（b）是否有某些“机械性的”限制影响了该设施适用的广泛型性。

（1）考虑通过普通参数来传递回调函数，而不是通过模板参数。非类型的模板参数某种程度上是不常用的，因为它意味着在编译期就严格地将类型确定下来，这几乎肯定不会带来太多的好处。换句话说，我们可以这么写：

```
template < class T >
class callback {
public:
  typedef void (T::*Func)();

  callback(T& t, Func func) : object(t), f(func) {}        // 绑定实际的对象
  void operator()() const { (object.*f)(); }               // 调用回调函数

private:
  T& object;
  Func f;
};
```

现在，注册的回调函数可以在运行期更换了，而且为 callback 类模板添加一个成员函数以允许用户改变当前回调对象所绑定到的成员函数也是极其简单的，而前一个版本的 callback 就做不到这一点。

262

准则：将非类型模板参数换成普通的函数参数通常是个好主意，除非它们确实应是模板参数[1]。

（2）使其能被放进容器内。如果一个程序想要将某个回调函数对象保留到以后再调用，它很可能会保留多个这类对象。如果它想要将这些回调对象都放进容器（例如 vector、list）中该怎么办呢？就目前的 callback 类来说这是不可能的，因为它目前尚不支持赋值，即不支持 operator=。为什么不支持呢？原因是它有一个引用型的成员，一旦该引用在构造期被绑定到了某个对象，那么以后就再也不可能将它重新绑定到其他对象。

然而，指针却没有这一限制，你给它什么它就指向什么。本例中让 callback 具有一个指针成员是完全安全的，而且这样一来我们就可以利用编译器默认生成的复制构造函数和复制赋值操作符了：

```
template < class T >
class callback {
public:
  typedef void (T::*Func)();

  callback(T& t, Func func) : object(&t), f(func) {}          // 绑定实际的对象
  void operator()() const { (object->*f)(); }                 // 调用回调函数

private:
  T* object;
  Func f;
};
```

这么一来，我们就可以创建出一个 list<callback<Widget> >，让其中的每个 callback<Widget>都指向不同的 Widget 成员函数。而倘若使用原来的 callback，我们就只能创建 list<callback<Widget,&Widget::SomeFunc>>;其中每个 callback 调用的都是同一个成员函数。

准则：尽量让你的对象与容器相兼容。特别是，一个对象要想能够被放进标准容器中，就必须是可赋值的。

“不过等一下”，你可能还会问：“如果可以创建一个那样的 list，那为什么不能创建一个包含各种不同类型的对象的成员回调函数的 list 呢？这样我可以将它们全部记录在一个 list 中，并在需要时回调它们。”的确，这是可以实现的，只需添加一个基类即可。

（3）加入多态能力：为 callback 类模板提供一个基类。如果想要让读者能够创建 list<call-

1. 换句话说，除非它们确实必须在编译期就确定下来。

backbase*>（或者，更酷一点的 list<shared_ptr<callbackbase> >），只需提供下面这个简单的基类： 263

```
class callbackbase {
public:
  virtual void operator()() const { };
  virtual ~callbackbase() = 0;
};

callbackbase::~callbackbase() { }

template < class T >
class callback : public callbackbase {
public:
  typedef void (T::*Func)();

  callback(T& t, Func func) : object(&t), f(func) {}          // 绑定到实际对象
  void operator()() const { (object->*f)(); }              // 调用回调函数

private:
  T* object;
  Func f;
};
```

这么一来，只要你愿意，随时都可以创建一个 list<callbackbase*>并多态地调用其各元素的 operator()。当然，一个 list<shared_ptr<callback> >应该更好一点参见[Sutter02b]。

注意，增加一个基类实质上是一种折中[1]，只不过付出的代价很少：当用基类接口触发**派生类中**的回调函数时，增加了虚函数转发的开销。这一开销也只有当使用基类接口时才会产生。不需要使用基类接口的代码不用承受这一开销。

准则：考虑为类（模板）加上多态能力，这样类模板的不同实例就可以被替换使用了，不过前提是这一点对类模板来说的确有意义。如果是这样的话，做法是为类模板提供一个基类，这样类模板的任何实例就都会继承自该基类了。

（4）（惯用法、折中）可以为 callback 类模板增加一个辅助进行类型推导的 make_callback() 函数。因为用户在用了 callback 类一阵子之后，可能会发觉每次在创建 callback 的临时对象时都要给出模板参数比较麻烦，见下面的代码： 264

```
list< callback< Widget > > l;
l.push_back(callback<Widget>(w, &Widget::SomeFunc));
```

为什么非得写两次 Widget 呢？难道编译器不知道吗？呃，的确不知道。不过，在只需要像这样一个临时对象的场合，我们可以帮助编译器获知这一信息。换种做法，我们也可以提供一个辅助

1. 效率与灵活性间的折中。——译者注

函数 make_callback(...)，这么一来用户就只需要写：

```
list< callback< Widget > > l;
l.push_back( make_callback( w, &Widget::SomeFunc ));
```

make_callback()与标准的 std::make_pair()的工作方式类似。make_callback()辅助函数应当是一个函数模板，因为只有这样编译器才能够推导出参数类型。下面就是 make_callback()的定义：

```
template<typename T >
callback<T> make_callback(T& t, void (T::*f) ()) {
  return callback<T>(t, f);
}
```

（5）（折中）增加对其他签名类型的回调函数的支持。我把这个最烫手的山芋留到最后来解决。因为除了 void (T::*F)()之外世界上还有很多种类型的函数！

准则：不要限制模板，避免对特定的类型（或不那么一般性的类型）进行硬编码。

如果要求所有的回调函数都具有 void (T::*)()类型就足够了，那我写到这可以算是功德圆满了。无缘无故地复杂化一个设计是毫无意义的，如果想要允许 callback 适应更多类型的回调函数，我们就得将现在的设计复杂化。

我并不打算列出所有的代码，因为那相当冗长（如果你真想看看这种重复性的编码，或者如果你患有失眠症，你可以去看看像[Alexandresuc01]这样的书或有关的文章，里面有类似的例子）。我打算做的是简要地列出你需要支持的主要特性以及如何支持它们：

首先，是 const 成员函数。支持 const 成员函数的最简单办法是提供另一个使用 const 签名类型的回调类，并记得让它接受/保存目标对象的 const 引用或指针。

其次是非 void 类型的返回值。允许返回类型可变的最简单途径是添加另一个模板参数。

265 其三，支持有参的回调函数。再一次，这可以通过添加模板参数来予以解决，不过别忘了给 operator()添加相应函数参数，而且要细心一点。

记住为每种可能的调用参数个数添加新的模板。

唉，代码膨胀，而且不得不人为地为 callback 类支持的回调函数的参数个数设一个上限。或许在未来的 C++0x 中会看到一个类似可变参数列表的模板特性，从而解决这个问题，但现在不能。

小结

将所有这些结合到一起，并进行一些纯粹语法上的调整（诸如一致使用 typename，使用我较喜欢的命名习惯和空格习惯等），得到如下的代码：

```
class CallbackBase {
public:
```

```
  virtual void operator()() const { };
  virtual ~CallbackBase() = 0;
};

CallbackBase::~CallbackBase() { }

template<typename T>
class Callback : public CallbackBase {
public:
  typedef void (T::*F)();

  Callback(T& t, F f)       : t_(&t), f_(f) { }
  void operator()() const { (t_->*f_)(); }

private:
  T* t_;
  F  f_;
};

template<typename T>
Callback<T> make_callback(T& t, void (T::*f) ()) {
  return Callback<T>(t, f);
}
```

266

第 36 条 构造式 union

难度：4

别误会，本条并不是要教你七拼八凑整出一个什么东西来，而是要讨论如何在“好的”跟“坏的”之间进行抉择，良好的动机是怎样最终误入歧途，以及在背后进行破坏性的举动会带来什么样的后果。一句话，本条讨论的是如何克服C++中不能将构造的对象放入union中这一限制的。

JG问题

1. 什么是 union，它们存在的目的是什么？

2. 哪些类型的对象不可以作为 union 的成员？为什么存在这一限制？请解释。

Guru问题

3. [Manley02]中提到了编写一门脚本语言的动机案例：假设你想要语言支持一个单一的类型，其变量在不同时候可以分别持有整型、string 或 list。而创建一个 union {int i;list<int> l;string s;} 又根本行不通（原因见问题 1 和问题 2）。下面的代码给出了一个弥补方案，该方案试图支持将任何类型放进 union 中的能力（原文中有更为详细的解释）。

评价这段代码，并指出其中：

（a）存在着哪些机械性的错误，如无效语法或不可移植的习惯等。

（b）可以进行哪些风格上的改进以便改善代码清晰性、可复用性以及可维护性。

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

#define max(a,b) (a)>(b)?(a):(b)

typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
   MYUNION() : currtype(NONE) {}
   ~MYUNION() {cleanup();}

   enum uniontype {NONE,_INT,_LIST,_STRING};
   uniontype currtype;
   inline int& getint();
   inline LIST& getlist();
   inline STRING& getstring();
```

267

```
protected:
  union {
    int i;
    unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
  } U;

  void cleanup();
};

inline int& MYUNION::getint()
{
  if(currtype==_INT) {
    return U.i;
  } else {
    cleanup();
    currtype=_INT;
    return U.i;
  } // else
}

inline LIST& MYUNION::getlist()
{
  if(currtype==_LIST) {
    return *(reinterpret_cast<LIST*>(U.buff));
  } else {
    cleanup();
    LIST* ptype = new(U.buff) LIST();
    currtype=_LIST;
    return *ptype;
  } // else
}

inline STRING& MYUNION::getstring()
{
  if(currtype==_STRING) {
    return *(reinterpret_cast<STRING*>(U.buff));
  } else {
    cleanup();
    STRING* ptype = new(U.buff) STRING();
    currtype=_STRING;
    return *ptype;
  } // else
}
void MYUNION::cleanup()
{
  switch(currtype) {
```

268

```
  case _LIST: {
  LIST& ptype = getlist();
  ptype.~LIST();
  break;
 } // case
 case _STRING: {
  STRING& ptype = getstring();
  ptype.~STRING();
  break;
  } // case
  default: break;
 } // switch
  currtype=NONE;
 }
```

4. 请想出一个更好的办法来实现泛化的 variant[1]类型，并对在实现过程中作出的任何折中进行评论和解释。

解决方案

再论union

1. 什么是 union，它们存在的目的是什么？

union 允许多个对象（不管是类还是内建类型）占用内存中的同一块空间。例如：

```
// 示例36-1
//
union U {
 int i;
 float f;
};

U u;

u.i = 42;                          //没问题，现在u.i处于活跃状态
std::cout << u.i << std::endl;

u.f = 3.14f;                       //没问题，现在u.f处于活跃状态
std::cout << 2 * u.f << std::endl;
```

269

然而，同一时刻只能有其中一个类型处于活跃状态，因为毕竟它们共享同一块内存，这块内存在某一特定时刻只能被其中一个对象占据。另外 union 也只支持某些特定的类型，这就把我们带入了下一个问题：

1. variant 即“可变的”意思，具体的说就是，variant 类型的变量的实际类型可以在运行期变化。参见 boost::variant。由于其特殊性，故保留不译。——译者注

2. 哪些类型的对象不可以作为 union 的成员？为什么存在这一限制？请解释。

下面来自 C++标准：

具有非平凡（non-trivial）构造函数、非平凡复制构造函数、非平凡析构函数或非平凡复制赋值操作符的类的对象和数组不可以作为 union 的成员。

简单地说，一个类类型要想能被放入 union，就必须满足如下要求：

- 只可以有编译器生成的构造函数、析构函数以及复制赋值操作符。
- 无虚函数或虚基类。
- 其所有基类和非静态成员（或数组）都满足以上两点要求。

就这些。不过这些已经排除了相当一部分类型。

union 是从 C 继承来的。C 语言在效率方面以及对接近硬件的底层编程的支持方面有着强大的传统，这一传统在 C++中得到了保留，这也正是 union 得以存在于 C++中的原因。另一方面，C 语言传统上对面向对象没有任何语言级别的支持，因而自然也就不支持具有构造函数、析构函数和用户自定义的复制构造函数的类类型。然而 C++是支持这些的，正因如此 C++才需要定义当这些（对 C 来说是）新型的类型碰到老式的 union 时哪些做法是有意义的以及怎样才能不违反 C++对象模型（包括 C++的对象生命期保证）。

如果 C++对 union 不加任何限制，一些糟糕的事情就可能乘虚而入。例如，考虑如果下面的代码被语言允许会发生什么：

```
// 示例36-2: 并非符合标准的C++代码，不过暂且假设它能够通过，看看会发生什么？
//
void f() {
  union IllegalImmoralAndFattening {
    std::string s;
    std::auto_ptr<int> p;
  };

  IllegalImmoralAndFattening iiaf;

  iiaf.s = "Hello, world";          // s的构造函数被调用了吗？
  iiaf.p = new int(4);              // p的构造函数被调用了吗？
}    // s会被析构吗?s应当在这里被析构吗？p会被析构吗？p应当在这里被析构吗？
```

270

正如注释中指出的，如果允许这样的代码就会带来严重的问题。当然我们可以对语言规则进行一些细致的调整，但那顶多只能修补其中的部分问题，况且考虑到避免使语言进一步复杂化，所以干脆禁止这些有问题或可能滋生问题的操作。

但不要认为 union 只是早期残留下来的东西。由于 union 支持数据的重叠式存放，而这一能力在 C++乃至现代编程领域仍然是有用的，故而 union 在节省空间方面或许可以说是最有用的特性。例如，C++标准库的某些最高级的实现就利用了 union 来实现“小型串优化”，“小型串优化”

是一项很好的优化方案，它复用了 string 对象本身内部的内存。下面介绍一下它的理念：对于大型串来说，string 对象内部的空间存放的是指向动态分配的缓冲区的指针以及一些系统维护信息（如缓冲区大小）；然而，对于小型串，这些空间被用来直接存放串本身的内容，完全避免了动态内存分配。参见 *More Exceptional C++* [Sutter02]的第 13 条至第 16 条中更多关于小型串优化技术（并相当深入地讨论了串优化的积极和消极的方面）的讨论，也可参见[Meyers01]中对目前的商业 std::string 实现的探讨。

剖析构造式union

3. [Manley02]中提到了编写一门脚本语言的动机案例：假设你想要语言支持一个单一的类型，其变量在不同时候可以分别持有整型、string 或 list。而创建一个 union {int i;list<int> l;string s;} 又根本行不通（原因见问题 1 和问题 2）。下面的代码给出了一个弥补方案，该方案试图支持将任何类型放进 union 中的能力（原文中有更为详细的解释）。

从积极的角度来看，这篇文章解决了一个实际的问题，而且为了提供一个好的解决方案显然花了不少工夫。然而遗憾的是，虽然大家的初衷都是好的，但很多程序员最后都误入了歧途。

该方案的设计和编码中的问题仍然可以归为三大类：合法性、安全性以及“道德规范”。

评价这段代码，并指出其中：

（a）存在着哪些机械性的错误，如无效语法或不可移植的习惯等。

（b）可以进行哪些风格上的改进以便改善代码清晰性、可复用性以及可维护性。

第一个需要作出的总体评价就是：该方案代码背后的基本理念在标准 C++中是不合法的。它的作者在文章中曾总结了它的关键理念：

271 关键理念是声明一个原始内存缓冲区而不是对象成员[该缓冲区并非动态分配的，而是一个成员 char 数组，它使类可以假装具有 union 的行为]，并通过定位（in-place）构造随时随地创建出所需的对象。——[Manley02]

这个想法很常见，但遗憾的是它是不可靠的。

分配某个类型的缓冲区并使用强制转换来将另一个类型的对象放进来拿出去，这种做法是不符合标准且不可移植的，因为非动态分配的缓冲区（即不是由 new 或 malloc()分配的缓冲区）除了保证能满足其本身类型的对齐要求之外，并不能保证对于其他类型也是正确对齐的。即便这种做法碰巧在某个人使用的编译器上对于某几个特定的类型能够工作，也不能保证对于其他类型（或者同一编译器的下一个版本中的同一类型）也能够工作。*Exceptional C++* [Sutter00]第 30 条对此进行了更为详细的讨论，此外还包括一些直接相关的讨论，特别是其中一段题为“不顾后果的修改和优化，以及为什么说它们是有害的”的讨论。也可以参见[Alexandrescu02]中关于对齐的讨论。

标准委员会正在考虑往 C++0x 中添加有关对齐的辅助设施，尤其是为了让这种依赖于对齐的技术实现起来更容易，但那些都是以后的事。眼下为了让该方案（就算是有时候）能够适度可

靠地工作，需要采取如下的办法之一：

- 借助于 max_align 手法（参见[Manley02]，其中以脚注介绍了 max_align 手法，或者也可以在 Google 上搜 max_align）。
- 借助于非标准扩展（例如 Gnu 的__alignof__）来让该方案能够在支持这类扩展的编译器上可靠地工作（虽说 Gnu 提供了一个 ALIGNOF 宏，该宏的意图是能够在其他编译器上比较可靠地工作，然而它毫无疑问也是借助于一些 hack 技巧来实现的，其能否正确工作依赖于编译器特定的对象布局方式以及基于 offsetof[1]查询作出的猜测，虽说这个猜测通常是准确的，但标准对此并没有任何保证）。

可以通过（使用 malloc()或 new）动态分配数组（缓冲区）来绕过这个问题，因为 malloc()和 new 保证它们分配的 char 缓冲区对于任何类型的对象都是恰当对齐的，只不过这个动态分配的想法仍然是个糟主意（它同样不是类型安全的），更何况原作者写作该类的动机之一就是效率，而采用动态分配的性能开销则抵消了这个好处。一个正确的替代方案是使用 boost::any（见下文），boost::any 也引入了类似的分配/间接层额外开销，但至少它是正确且安全的。我们待会再来讨论。

如果试图对抗语言规则，甚至试图让语言按照你的意愿来行事，那么你得注意了，这种做法通常是有问题的，应当打上一个大大的红色惊叹号。我就曾在刚才提到的 *Exceptional C++* [Sutter00]中关于不计后果的优化的讨论中严厉谴责了一个类似的技巧。的确，有些情况下使用某些明确知道不可移植的结构也是合理的，只要它在某些特定场合下能够工作即可（在本例中对应的就是 max_align 手法）。但即便如此，我还会争辩说这种做法应当被密切注意，并进一步指出它在那些意图被广泛使用的代码中是没有立足之地的。 272

进入代码

现在让我们来考虑代码：

```
#include <list>
#include <string>
#include <iostream>
using namespace std;
```

总是包含必要的头文件。因为下面的代码中用到了 new，所以需要#include <new>（注意：#include <iostream>是必要的，因为在代码的后半部分有一个测试用例会用到输出流）。

```
#define max(a,b) (a)>(b)?(a):(b)

typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
  MYUNION() : currtype(NONE) {}
```

1. offsetof()是 C 中一个著名的宏，其用途是求出成员在结构中的偏移。例如假设 struct S 拥有一个成员 m，那么 offsetof(S,m)得到的就是 m 在 S 中的偏移。——译者注

```
~MYUNION() {cleanup();}
```

以上代码中的一个经典的机械错误是 MYUNION 的复制是不安全的，因为编写者忘记了提供合适的复制构造函数和复制赋值操作符。

MYUNION 打算用构造函数和析构函数来做一些事情，因此它显式提供了这两个函数。这本身并没有任何问题，但问题是仅仅这两个函数是不够的，因为 MYUNION 所玩的“游戏”的性质决定了复制构造函数和复制赋值操作符也得参与进来，而实际上原来的代码中这两个函数是没有被显式提供出来的。这很糟糕，因为编译器隐式生成的默认复制操作根本无法胜任：它们会按位复制其中的 char 数组缓冲区，这极有可能带来极不愉快的后果，大多数情况下会直接导致内存错误。下面的例子展示了其危害：

```
// 示例36-3: MYUNION的复制行为是不安全的
//
{
  MYUNION u1, u2;
  u1.getstring() = "Hello, world";
  u2 = u1;                                  // 按位复制u1至u2
}// 哎呀！串的内容被两次delete[1]（假设按位复制是有意义的）
```

273

准则：遵守“大三法则”[Cline99]：如果一个类需要自定义的复制构造函数、复制赋值操作符或析构函数中的任一个，那么它往往同时需要这三者。

从这个经典的错误往下，我们又遇到了一对经典的风格错误：

```
enum uniontype {NONE,_INT,_LIST,_STRING};
uniontype currtype;

inline int& getint();
inline LIST& getlist();
inline STRING& getstring();
```

以上代码中存在两个风格上的错误。首先，该结构是不可复用的，因为它对特定的类型进行了硬编码。实际上，提出这个方案的作者建议每次需要的时候都通过手动编写这么一个结构。其次，虽说该结构也有着有限的、意料之中的用途，但它的可扩展性或可维护性可不大好。我们在了解了更多上下文之后会再回到这个问题。

准则：尽量避免硬编码，因为它往往会（不必要地）令代码变得更脆弱，同时还会限制其灵活性。

另外还有两个问题：首先是 currtype 没理由放在公用区段，这违背了良好的封装准则，意味着任何用户都可以自由地对它进行篡改，尽管某些改动可能未必是有心的。第二个问题则关系到

1. 这里发生的按位复制会使两个 MYUNION 的 string 对象中的指针指向同一块内存（其中存放着“Hello, world”），即所谓的浅复制问题，当这两个 string 析构的时候就会导致这块内存被释放两次。——译者注

枚举中使用的名字，后面我会单独用一节来讨论这个问题。

protected:

这是一个错误：内部实现应当为私有而非受保护的。将它们放在受保护区段的惟一理由是让派生类可以访问到它们，但 MYUNION 最好还是不要有什么派生类，因为从 MYUNION 进行派生是不安全的，原因不仅是因为它在内部玩的“不足为外人道”的把戏，还有一个原因就是它缺少虚析构函数。

准则：总是将所有数据成员放在私有区段。惟一的例外是C风格结构，后者的目的并非封装，其所有数据成员都是公用的。

274

```
  union {
    int i;
    unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
  } U;

  void cleanup();
};
```

主类的定义就这些。接下来我们考虑三个平行的访问函数：

```
inline int& MYUNION::getint()
{
  if(currtype==_INT) {
      return U.i;
  } else {
      cleanup();
      currtype=_INT;
      return U.i;
  } // else
}

inline LIST& MYUNION::getlist()
{
  if(currtype==_LIST) {
      return *(reinterpret_cast<LIST*>(U.buff));
  } else {
      cleanup();
      LIST* ptype = new(U.buff) LIST();
      currtype=_LIST;
      return *ptype;
  } // else
}

inline STRING& MYUNION::getstring()
{
```

```
    if(currtype==_STRING) {
      return *(reinterpret_cast<STRING*>(U.buff));
    } else {
      cleanup();
      STRING* ptype = new(U.buff) STRING();
      currtype=_STRING;
      return *ptype;
    } // else
  }
```

275 一个小缺点：注释//else 什么用场也不起。很遗憾，代码中仅有的注释也是多余的。

准则：（只）写有用的注释。注释不应该表达代码已经表达过的意思，而应该解释代码，以及解释为什么要这样编写这段代码。

更为严肃的，下面还有 3 个主要问题。首先，函数的编写是不对称的：getlist()或 getstring()的第一次使用会默认构造一个相应的对象并将其返回，而 getint()的第一次使用却只得到一个未初始化的对象。如果这是故意而为之的，是为了符合 int 变量不初始化这一通常的语义，那么 MYUNION 的这一性质或行为就应当在文档中加以注明。但由于实际上这并非有意为之，因此 int 对象也应当被初始化。例如，如果调用方调用了 getint()并试图将得到的值（未初始化的）作一份复制，那么结果就是未定义的——并非所有的平台都支持复制任意（无效的）int 值，有些平台会在运行期拒绝执行相应的指令。

第二个主要问题是用户无法保证 const 正确地使用这几个函数。如果确实打算这么编写这几个函数，那么至少为它们中的每一个都提供一个 const 的重载；这些 const 版本与它们的非 const 版本返回同样的东西，只不过加上了 const 修饰。

准则：确保const正确性。

第三个主要问题就是这种做法是脆弱的，在变化面前显得弱不禁风。它依赖于类型 switch，因而在添加或删除它所支持的类型时就得保持所有成员函数的同步改变，这相当麻烦，而且很容易出错。

当你读到这儿的时候，请停下来考虑一个问题：如果想要为 MYUNION 添加对某个新类的支持，该对已经发布的代码作哪些改变？请尽量将需要做的全部罗列出来。

……

怎么样？退缩了吗？好吧，下面就是我能给出的一个清单。为了添加对某个新类型的支持，需要记住做下面的事情：

- 添加一个新的枚举值；
- 添加一个新的成员函数用于访问该类型的对象。

- 更新 cleanup()函数，让它能够安全地销毁这个新类型的对象；
- 将这个新类型纳入 max()计算，以确保 buff 的大小足以容纳这个新类型的对象。

要是你忘记了上面的任何一点乃至几点，唔……这正好说明了该方案的维护和扩展有多困难。 276

再往下，我们就到了最后一个函数：

```
void MYUNION::cleanup()
{
 switch(currtype) {
  case _LIST: {
   LIST& ptype = getlist();
   ptype.~LIST();
   break;
  } // case
  case _STRING: {
   STRING& ptype = getstring();
   ptype.~STRING();
   break;
  } // case
  default: break;
 } // switch
 currtype=NONE;
}
```

同样，这段代码中的注释毫无用处：//case 和//switch 没有为代码增添任何信息，很遗憾就连仅有的这点注释也要删掉。如果一段注释的惟一作用是分散阅读者的注意力，那还不如不要这段注释。

但这段代码还有一个更大的问题：switch 语句中简单地使用了 default:break，较好的做法是列出所有的类型（包括 int），如果类型未知就会引发一个逻辑错误——可以采用断言，也可以抛出一个 std::logic_error(...)。

再次申明，类型 switch 纯粹是有害的。在 Google 上搜 switch C++ Dewhurst 会找到关于这个话题的各式各样有趣的讨论，包括[Dewhurst02]。如果需要更多的证据来说服你的同事避免使用类型 switch，可以根据 Google 给出的线索找到更多的详细讨论。

准则：避免使用类型switch；尽量寻求类型安全性。

危险的名字

还剩下最后一个机械性问题没有提到。该问题第一次出现是在下面这行代码中：

```
enum uniontype {NONE,_INT,_LIST,_STRING};
```

277

永远、永远不要创建以下划线开头或其中包含有双下划线的名字，它们是专门为编译器和标准库

厂商保留的，之所以是“专门”是为了让编译器和标准库能够有专属于它们自己的名字，这样它们在使用这类名字时就不会影响到代码。所以倘若侵犯了它们的名字，往往结果是它们的名字会“反咬你一口”！[1]

别停！接着往下读！你可能以前看到过这条建议，甚至可能在我的文章中读到过这样的建议。你甚至可能对它感到厌烦了，正准备忽略本节的剩余部分跳到下一节去。如果是这样，下面这段讨论就是为你准备的，因为这条建议并非纸上谈兵，在下面的代码中你会看到它带来了什么样的危害。

刚才给出的枚举定义在我试过的大多数编译器上都通过了编译：Borland 5.5、Comeau 4.3.0.1、gcc 2.95.3 / 3.1.1 / 3.4、Intel 7.0 以及 Microsoft Visual C++ 6.0 至 8.0 (2005) beta 版。但在其中的两个编译器（Metrowerks CodeWarrior 8.2 和 EDG 3.0.1，使用的是 Dinkumware4.0 标准库）上却导致了可怕的错误。

在 Metrowerks CodeWarrior 8 上，这短短的一行代码首先就惹来了 52 个（我并没有打错数字）错误，长达 225 行（再一次，仍然不是打字错误）的错误消息以下面这段诊断信息开始（它直接指向其中一个逗号）：

```
### mwcc Compiler:
#   File: 36.cpp
# ----------
#    17:    enum uniontype {NONE,_INT,_LIST,_STRING};
#   Error:
#   identifier expected
### mwcc Compiler:
#    18:   uniontype currtype;
#   Error:  ^^^^^^^^
#   declaration syntax error
```

接下来洋洋洒洒又是 52 个错误、215 行错误信息。对于第二个以及后续的错误，很明显我们应当忽略它们，因为它们只不过是从第一个错误衍生来的——由于 uniontype 没有被成功地定义，因此后面广泛使用了 uniontype 的代码也就理所当然地吃了闭门羹。

但问题是 uniontype 的定义又怎么了呢？编译器指向的那个逗号看上去当然位于合适的位置啊，不是吗？在它前面明明有一个标识符不是吗？当我们看到 Metrowerks 编译器给出的预编译输出后，一切都真相大白了。在略去若干行之后，我们看到了预编译的结果：

278

```
enum uniontype {NONE,_INT,, };
```

啊哈！这是非法的 C++代码，编译器当然有理由抱怨其中的第三个逗号放得不是地方，因为在它前面根本没有任何标识符。

1. 说得更细一点，规则就是：任何其内部包含了双下划线的名字（如 like_this）或者以下划线后跟大写字母开头的名字（如_LikeThis）都是保留的。如果你喜欢，可以记住这条规则，不过更容易的做法是一概不用前导下划线和含有双下划线的名字。

但_LIST 和_STRING 究竟到哪去了呢？你猜怎么着，它们被“饥饿的”编译器给“消化”掉了。因为凑巧的是 Metrowerks 的实现定义了_LIST 和_STRING 宏[1]，因此代码中的这两个名字就被替换掉了，这一行为是完全合法的，因为标准本来就允许实现拥有_Names 以及__names 这样的名字。

那么，也就是说 Metrowerks 的实现碰巧将_LIST 和_STRING“吞”掉了，这便合理地解释了发生在 Metrowerks 身上的“神秘事故”。但 EDG/Dinkumware 实现呢？你自己去判断：

```
"1.cpp", line 17: error: trailing comma is nonstandard
    enum uniontype {NONE,_INT,_LIST,_STRING};
                                    ^

"1.cpp", line 58: error: expected an expression
    if(currtype==_STRING) {
                          ^

"1.cpp", line 63: error: expected an expression
        currtype=_STRING;
                         ^

"1.cpp", line 76: error: expected an expression
        case _STRING: {
                      ^

4 errors detected in the compilation of "36.cpp".
```

这一次，不用生成和考察该文件的预编译后版本，也能够看出到底发生了什么：编译器表现得就好像_STRING 根本不在那儿一样，这是因为这个名字让预处理器给“吃”了。

我希望所有这些能够使你相信，“不使用像_Names like__this 这样的名字”一说绝非空穴来风或纸上谈兵。它的确是个实实在在的问题，因为命名上的限制会直接影响到代码与编译器和标准库作者之间的关系。如果你试图闯进它们的领地，你可能会毫发无损的回来，但也可能会遍体鳞伤。 279

C++的疆域是宽广而清晰的，它允许编写各种各样奇妙而灵活的代码，允许你跟随一个开发者内心的意愿信步闲庭。当然它也允许在标准名字空间之外任意选择所中意的名字。只不过在名字方面，C++同样保留了一块“闲人免进”的领地，这块领地由散发着不友好的青光的铁丝网围住，任何擅闯者会得到警告：“私自闯入者后果自负”。上面的例子就是不顾警告的后果！

准则：永远不要使用保留名字，即以前导下划线开头或含有双下划线的名字。它们是语言为编译器和标准库实现保留的。

寻找一个更好的做法：boost::any

4. 请想出一个更好的办法来实现泛化的 variant 类型，并对在实现过程中作出的任何折中进行评论和解释。

---

1. 如#define _LIST。——译者注

[Manley02]的原文这样写道：

你可能想要实现这样一门脚本语言，它支持单个变量类型，该类型的变量既可以持有一个整型，又可以持有一个字符串或一个链表等。——[Manley02]

这没错，而且到目前为止也没有疑义。但文中接着写道：

欲实现这样一个组合式类型，union 是不二之选。——[Manley02]

恰恰相反，文章中给出的示例代码从相当细节的层面上说明了为什么 union 根本不能胜任这个任务。

但如果不是 union，又能是什么呢？欲实现这样一个 variant 类型，一个不错的候选是 Boost 库中的 boost::any，以及与它相关的众多 any_cast[1]。有趣的是，一个完全泛化的 any 的完整实现所需的代码量与上面展示的 MYUNION 类的代码量差不多，MYUNION 的硬编码性质决定了它只能支持一组特定的类型（本例中是 int、list<int>和 string），而 any 则是完全泛化的、可扩展的，而且还是类型安全的。

然而，boost::any 毕竟还是进行了一次折中，即动态分配。boost::any 并没有试图通过避免动态内存分配来获得潜在的效率提升，而后者正是原文所提到的方案动机之一。

280

同样值得注意的是，如果开始的解决方案使用单个动态分配的缓冲区来实现，boost::any 的额外开销仍然还是比它大，这是因为 boost::any 每当包含的类型改变时都会重新申请内存，而 MYUNION（如果使用动态分配的话）则不然，它在整个生命期内只进行一次动态分配，以后只不过是重复使用分配得来的内存而已。

原文的测试用例如果改用 boost::any，则看起来会是下面这样（原来使用 MYUNION 的代码则被放在后面的注释中以供对照）。

```
any u;              // 原先是: MYUNION u;
```

这么一来就不用手写结构了，而是只需直接使用 boost::any 即可。注意，boost::any 只是一个普通类而已，并非类模板。

```
// 把u当成整型来访问
u = 12345;          // 原先是: u.getint() = 12345;
```

赋值操作则更是显示出了 boost::any 较之 MYUNION 更为自然的语法。

```
cout << "int="<< any_cast<int>(u) << endl;      // 或者只写int(u)
                                // 原先是: cout << "int="<< u.getint() << endl;
```

在 int(u)与 any_cast<int>(u)中我更喜欢后者，因为它更一般化（包括它是一个非成员的自由函数），从 C++风格的角度来看也更为自然。如果已经知道确切的类型，当然也可以使用更为简洁的转换形式，即 int(u)这样的 C 风格转换。另一方面，MYUNION 的 get_类型则显得较为脆弱，难于编写和维护等等。

1. [Hyslop01]中有更多关于这一设施的讨论。

```
// 将u作为一个std::list<int>来访问
u = list<int>();
list<int>& l = *any_cast<list<int> >(&u);        // 原先是: LIST& list = u.getlist();
l.push_back(5);                                   // 同样: list.push_back(5);
l.push_back(10);                                  // 同样: list.push_back(10);
l.push_back(15);                                  // 同样: list.push_back(15);
```

我认为 any_cast 还可以再改进，以便我们更容易地通过它来获得引用，这样也还算过得去。（此外，当作用域中模板的名字是 list 时，我不建议使用 list 作为变量名，这太可能导致表达式的二义性）。

到目前为止，我们进行了一些可读性和类型支持性/类型安全性方面的改进。剩下来的区别就更小了：

```
list<int>::iterator it = l.begin();                // 原先为: LIST::iterator it = list.begin();

while(it != l.end()) {
  cout << "list item="<< *(it) << endl;
  it++;
} // while
```

281

几乎没什么改变。我没有删掉源代码中的//while 注释，因为它与所进行的比较没有关系。

```
// 将u当成std::string来访问
u = string("Hello world!");                        // 原先为: STRING& str = u.getstring();
                                                   //         str = "Hello world!";
```

再一次，使用 boost::any 的代码版本比原先的版本要简洁，但也只简洁一点而已。

```
cout << "string='"<< any_cast<string>(u) << "'"<< endl; //    或者 "string(u)"
                    // 原先为: cout << "string='"<< str.c_str() << "'"<< endl;
```

同上。

Alexandrescu的可识别Union

实际上，完全可能在遵循标准 C++的前提下实现一个满足最初目的（一是类型安全性，二是避免动态内存分配）的 variant 类。这像是 Andrei Alexandrescu 这种人喜欢解决的问题，尤其是考虑到在解决的过程中可能会涉及复杂的模板技术。[Alexandrescu02]就是明证，Alexandrescu 在这本书中描述了可识别 union（或称 Variant）方案，看起来：

- 可识别 union 从功能上等同于一个 union（Alexandrescu 总是喜欢对付这类问题）。
- 可识别 union 具有上面所说的几点优势。（通过一些诡秘的模板技巧，我只引用[Alexandrescu02]中的一句话："你知道 union 也可以是一个模板吗?"）
- 他做到了。

简而言之，通过尽可能地探索并延伸语言的边界，Alexandrescu 的 Variant 类型已非常接近于一个真正可移植的解决方案。它只有一点细微的缺陷，但在实际中或许已经是足够可移植了，尽管

它已经超越了语言标准所能作出的保证的范畴。实际上它的主要问题在于（即便忽略对齐相关的问题）Variant 代码的复杂性和先进性，使得它的代码只能在非常少的编译器上通过，实际上我只在一个编译器上尝试通过。

Alexandrescu 的 Variant 方案的一个关键部分是试图将 max_align 泛化成一个可复用的库设施（而且依然是符合标准 C++的）。之所以需要这么一个方案就是为了特意解决 MYUNION 内缓冲区的对齐问题，借助于该方案就可以继续相对安全地使用非动态的 char 缓冲区了。Alexandrescu 利用模板元编程计算出一个安全的对齐值。那么他的做法是可移植的吗？他对这个问题的讨论是这样的：

282

即便有最好的 Align，以上的实现仍然并非百分之百对所有类型是可移植的。理论上人们可以实现一个遵从标准但不与可识别 union 合作的编译器，这是因为标准并没有保证所有用户自定义类型最终都具有某个 POD 类型的对齐。不过就算真有这种编译器，它也更像是个缺德的语言律师头脑中妄想的产物，而不是一个实际的语言实现。

[……] 可移植地计算对齐要求很难做到，但仍是可行的，只不过永远达不到百分百的可移植性。——[Alexandrescu02]

Alexandrescu 的实现还有其他关键特性，特别是一个接受 typelist 作为模板参数的 union 模板（typelist 用于指出 union 中存放哪些类型的对象）以及可扩展的 visit 模式支持，另外还有为了避免在访问包含的对象时耗费一个额外的间接转发而使用的“伪虚函数表”技术。这些技术比 boost::any 更加重量级，不过从理论上仍然是可移植的。注意，只是“从理论上可移植”，我之所以这样说是因为 Andrei 在 *Modern C++ Design* [Alexandrescu01]中使用的技术大量运用了模板，以至于代码中出现了这样的注释：“在[Metrowerks、Microsoft、Gnu gcc1 等好几种流行的编译器上肯定会导致内部编译器错误”，而且其测试用例中也包含了一行被注释掉的代码，原因写道“下面这段代码在任何编译器上都不能工作”。

这正是 Variant 的主要缺点：对于它的实现代码，大多数现实中的编译器只能望洋兴叹，而且其代码虽说应当被严肃对待，但终究还是试验性的。我曾尝试用各种编译器（Borland 5.5; Comeau 4.3.0.1; EDG 3.0.1; gcc 2.95, 3.1.1、3.2; Intel 7.0; Metrowerks 8.2; Microsoft VC++ 6.0、7.0 (2002)、7.1 (2003)）来编译 Alexandrescu 的 Variant 代码。正如很多读者知道的，这些编译器中一些已经是非常强、非常符合标准的，然而它们在 Variant 面前却无一例外地败下阵来。

我试图对代码作一些小小的改动从而让它至少通过其中的某些编译器，然而我只成功地让它在 Microsoft VC++7.1(2003)上通过了，而在大多数编译器上甚至连指望都没有，因为它们对模板根本没有足够强的支持，因而在 Alexandrescu 的代码面前毫无招架之力（其中有些编译器会给出多得吓人的警告和错误——Intel 7.0 在编写它的 main.cpp 时给出了令人吃惊的长达 430k（真的，几乎都有半兆）的诊断信息！

为了让 Variant 的代码在 Microsoft VC++ 7.1(2003)上顺利通过编译而不报错（尽管调到最高警告级别仍然会报一些窄转换警告），我不得不做了如下三个改动：

283

- 往 AlignedPOD 类中添加了一个原先缺少的 typename。
- 添加了一个 this->，以便使 ConverterTo<>::-Unit<>::DoVisit()成为依赖名字。
- 在几个头文件的末尾添加换行符，因为 C++标准有这一要求（有些符合标准的编译器并不太介意这一点，允许文件末尾缺少换行符，并以此作为一个扩展特性；而 VC++则更为严格一点）[1]。

正如[Manley02]的作者进一步对 Alexandrescu 设计中的折中进行的评价：

它没有使用动态内存，而且也避免了对齐问题和类型转换。然而遗憾的是我找不到哪个编译器是能够编译它的代码的，因此我无法将其性能与 myunion 和 any 进行比较。Alexandrescu 的方案需要 **9 个支持头文件**，共大约 80KB，这就带来了它自身的可维护性问题。——K. Manley, private communication

他说得一点没错。

这里我不打算再进一步详述 Andrei 的三篇相关的文章了，不过我强烈建议读者中有兴趣的人找来阅读一下。它们在网上就可以找到，请参考后面的参考书目。

准则：如果要表示一个variant类型，就目前而言，应当尽量使用boost::any（或某些同样简单的东西）[2]。

一旦使用的编译器在对模板的支持方面跟得上，标准在对对齐的支持方面也跟得上，而且 Variant 库也跟得上（成熟的实现），那就是时候考虑使用类似 Variant 的库来作为 union 的类型安全的替代版了。

小结

即便 MYUNION 的设计和实现有所不足，但其动机是现实的，而且值得考虑。我得感谢 Manley 花时间写了那篇文章并引起人们对这种 variant 类型的需要性的关注，同样感谢 Kevlin Henney 和 Andrei Alexandrescu 给出了他们各自的解决方案。他们的解决方案严格来说并非百分之百可移植且符合 C++标准的，尽管 Alexandrescu 为了使 Variant 符合标准作出了极大的努力，并且其实现也的确从理论上非常接近于可移植了，但在现实中离可移植仍然尚远，因为只有少得可怜的编译器能够处理它的代码中用到的高级模板技术。 284

就目前而言，类似 boost::any 的方案是满足要求的最佳候选。如果在某些特定的场合，测试表明确实需要类似 Alexandrescu 的 Variant 的具有高效及其他特性的方案，而且你有足够的时间，对模板也有一定程度的了解，或许可以试着编写你自己“精简版”的 Variant（只实现[Alexandrescu02]中提到的可以运用到你的实际情况中的想法）。 285

---

1. 感谢同事 Jeff Peil 指出[C++03]中有这个要求，具体在[C++03] 2.1/1 节：“如果一个非空源文件不以换行符结尾，或者结尾的换行符前面紧跟着一个反斜杠，结果就是未定义的。”
2. 实际上，boost 中已经加入了一个能在许多编译器上通过编译的 variant 实现，即 Boost.Variant。——译者注

第 37 条 分解 std::string 之一：概观 std::string

难度：3

我打算以一种特别的方式来总结“风格案例研究”部分：一个迷你的系列条款，针对C++标准库中的std::string。我们先来回顾一个重要的设计原则，然后对照着对标准string设施的设计进行一些概观。

JG问题

1. 何谓单片式设计，为什么这种设计是糟糕的？请解释。

Guru问题

2. 列出 std::basic_string 的所有成员函数。

解决方案

避免不恰当的单片式设计

1. 何谓单片式设计，为什么这种设计是糟糕的？请解释。

“monolithic”一词用来描述那些单个、巨大、不可分割的软件，就像一块巨大的顽石。这个词来源于“mono”（一个）和“lith”（石头），它生动地描述了这类代码笨重和不可分割的性质。

认为单个的重量级设施就能够完成所有事情通常是行不通的，毕竟，这个设施并不一定能够做得更多，相反往往做得更少，因为它越是复杂，其应用和适用性就越是狭窄。

特别地，如果一个类总是试图通过成员函数而不是非成员函数来提供功能，甚至在可以用非成员非友元函数来实现而且实现起来并不比成员函数差的情况下也固执地使用成员函数，这个类就掉进了所谓的“单片式设计陷阱”。

286

- （主要问题）它把可能是独立可用的功能“封锁”在了一个类的内部。有些操作除了可以被这个类使用外或许还可以用在其他类型身上，但由于已经被死死地捆绑在了某个特定类的内部，因此这种可能性也就丧失了，而如果使用非成员函数来实现同样的功能，该功能的适用范围就广多了。
- （次要问题）这种设计可能会阻碍你为一个类扩展出新功能。“但是等等！”有些人可能会说：“一个类的已有接口主要使用成员函数还是非成员函数并没多大区别，因为可以使用自己的新非成员函数来扩展它！”这种说法从技术上来说的确没错，不过忽略了一个重要的方面：如果该类的内建功能主要是由（或者仅由）成员函数提供的，从而将此定为该

类的惯用法，而扩展的非成员函数不可能遵循它的原始惯用法[1]，因而总是“二等公民”。该类用成员函数来提供它的功能其实就相当于给用户们进行了语义上的陈述，于是用户就会习惯于在使用该类时使用成员调用语法，而扩展功能却做不到这一点（难道真的希望用户常常查询文档，看看哪个函数是成员哪个不是吗？就连较好的设计往往也会发生这种情况）。

尽可能将泛型组件分解为一个个的小部件。

准则：尽可能遵循单一职守原则（不论是对于类还是对于函数）。
尽可能编写非成员非友元形式的函数。

本条的其余部分将会在上述第二条原则上作进一步的讨论。

string基础

2. 列出 std::string 所有的成员函数。

呃……这可是个相当长的列表。

算上构造函数，std::string 拥有不少于 103 个成员函数。真的。如果这还算不上是“单片式”设计，很难想像还有什么能够称作单片式设计。

设想自己处在地下洞窟中的地下湖边上，有一条被水淹没的通道通往下一个充满空气的洞窟，你准备潜入阴冷的水中，沿被水淹没的通道游过去……

现在，深吸一口气，跟随 ISO/IEC 14882:2003(E)[2]：

287

```
namespace std {
 template<class charT, class traits = char_traits<charT>,
       class Allocator = allocator<charT> >
 class basic_string {

  // ……一些typedef……

  explicit basic_string(const Allocator& a = Allocator());
  basic_string(const basic_string& str, size_type pos = 0,
                  size_type n = npos, const Allocator& a = Allocator());
  basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
  basic_string(const charT* s, const Allocator& a = Allocator());
  basic_string(size_type n, charT c, const Allocator& a = Allocator());
  template<class InputIterator>
   basic_string(InputIterator begin, InputIterator end,
                  const Allocator& a = Allocator());
```

1. 无法向一个现有类中添加成员函数。——译者注
2. 一本厚厚的大部头，也称 ISO C++标准[C++03]。

```
~basic_string();
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const;
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;

const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
void push_back(const charT);

basic_string& assign(const basic_string&);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
```

288

```
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);

basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c);
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
  void insert(iterator p, InputIterator first, InputIterator last);
```

你浮上水面换口气。别放弃，我们已经走了一半！现在深呼吸、屏住气，继续下潜：

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);

basic_string& replace(iterator i1, iterator i2, const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
template<class InputIterator>
  basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);

size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string<charT,traits,Allocator>&);

const charT* c_str() const;            // 显式
const charT* data() const;
allocator_type get_allocator() const;

size_type find (const basic_string& str, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const;
size_type rfind(const basic_string& str, size_type pos =    npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(const charT* s, size_type pos = npos) const;
```

289

```
    size_type rfind(charT c, size_type pos = npos) const;

    size_type find_first_of(const basic_string& str, size_type pos = 0) const;
    size_type find_first_of(const charT* s, size_type pos, size_type n) const;
    size_type find_first_of(const charT* s, size_type pos = 0) const;
    size_type find_first_of(charT c, size_type pos = 0) const;
    size_type find_last_of (const basic_string& str, size_type pos = npos) const;
    size_type find_last_of (const charT* s, size_type pos, size_type n) const;
    size_type find_last_of (const charT* s, size_type pos = npos) const;
    size_type find_last_of (charT c, size_type pos = npos) const;

    size_type find_first_not_of(const basic_string& str, size_type pos = 0) const;
    size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
    size_type find_first_not_of(const charT* s, size_type pos = 0) const;
    size_type find_first_not_of(charT c, size_type pos = 0) const;
    size_type find_last_not_of (const basic_string& str, size_type pos = npos) const;
    size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
    size_type find_last_not_of (const charT* s, size_type pos = npos) const;
    size_type find_last_not_of (charT c, size_type pos = npos) const;

    basic_string substr(size_type pos = 0, size_type n = npos) const;
    int compare(const basic_string& str) const;
    int compare(size_type pos1, size_type n1, const basic_string& str) const;
    int compare(size_type pos1, size_type n1, const basic_string& str,
                size_type pos2, size_type n2) const;
    int compare(const charT* s) const;
    int compare(size_type pos1, size_type n1,
                const charT* s, size_type n2 = npos) const;
  };
}
```

290

喔，103 个成员函数（模板）终于完全展现在我们面前。岩石的通道终于开始变宽，我们及时从另一个湖的湖面上浮了出来。有点狼狈，但相对于获得的经验来说是值得的。环顾新的洞窟，看看刚才的一番辛苦是否给我们带来了新的发现或者有用的东西。

小结

尽可能将泛型组件分解为一个个的小部件。

- 尽可能遵循单一职守原则（不论是对于类还是对于函数）。
- 尽可能编写非成员非友元形式的函数。

将所有这些成员函数都看上一遍值得吗？所幸的是辛苦的工作终于完成了，下面回头再看一眼刚才所完成的工作，下一条将利用我们刚才获得的信息。

291

第 38 条　分解 std::string 之二：重构 std::string　难度：5

把所有东西都搅和在一块或许对厨师是适合的，对我们这些类设计者来说则大不合适。下面就是一个例子，从中可以看出当设计演变成过度设计时会发生多么糟糕的事情。很遗憾的是，这个例子正是从我们身边的标准库中摘选出来的。

JG问题

1. std::string 的哪些成员函数必须是成员形式的？为什么？

Guru问题

2. std::string 的哪些成员函数应当为成员形式的？为什么？

3. 解释为什么 std::string 的 at()、clear()、empty()和 length() 这 4 个成员函数可以在不损失泛型性或可用性、且对 std::string 的接口的其他部分没有影响的前提下改写成非成员非友元形式。

解决方案

成员关系有它的好处，但也要付出代价

还记得上一条中的建议吗？“在可行的情况下，尽量将泛型组件分成小部件；尽可能遵循单一职守原则（不论是对于类还是对于函数）；尽可能编写非成员非友元形式的函数”。

例如，如果编写一个 string 类，并将查找、模式匹配以及标记化功能全都作为成员函数，就等于是将这些功能牢牢地捆绑在 string 上，它们再也不可能被复用到其他序列上（如果这番话令你对 basic_string 心生不愉快的话，很好）。从另一方面来说，一个实现了同样目标但是由若干独立可用的部件构成的设施通常是良好设计的产物。在这里所讨论的情况下，通常最好将算法跟容器分离开来，STL 大多数时候正是这么做的。

我（*Exceptional C++* [Sutter00]的第 31 条和 34 条）和 Scott Meyers（[Meyers00]）都曾在书中讨论过为什么某些非成员函数也是一个类型的接口的正式一分子，以及为什么应当尽量采用非成员非友元形式的函数，其中一条原因就是为了改善封装性。例如，Scott 在我引用的这篇文章的开头这样说道：

292

> 这么说吧，如果你写的函数既可以实现为成员函数又可以实现为非成员函数，那么应当尽量将它实现为非成员函数。因为这一决策能够改善类的封装性。是的，当考虑封装性的时候，应当考虑将函数实现为非成员形式。——[Meyers00]

因此，当考虑对 basic_string（或其他任何类）进行操作的函数时，应当尽可能使它们成为非成员非友元函数。因此，下面就是在考察 basic_string 的成员函数时需要考虑的几点问题：

- 如果一个函数必须为成员函数，就使它成为成员函数。那么，究竟哪些操作必须是成员形式的呢？一方面，这可能是因为 C++语言标准规定的（例如构造函数），另一方面，也有可能其功能决定了它必须是成员形式的（例如虚函数）。如果一个函数必须是成员函数，那么……唔……当然必须是，没什么好多说的。
- 如果一个函数需要有对类内部的访问权，那么最好将它实现为成员函数。哪些操作需要访问类的内部数据，且如果它们不是成员函数就得通过将它们声明为友元来给它们这个权限呢？这样的操作通常应当设为成员函数。注意，也有极个别的例外，例如需要允许左边的参数能够是其他类型的操作，此外还有一些像 operator<<这样的其签名不允许 *this 引用作为其第一个参数的操作符；而且即便是这些非友元函数也可以基于（可能是虚）成员函数来实现，不过这样做没多大意义，最好还是老老实实地（也最自然）将它们声明为友元。
- 在其他任何情况下，尽量写非成员非友元函数。哪些操作被写成非成员非友元函数后照样工作得好好的？这些操作就是，因而它们通常应当被写成非成员函数。将操作写成非成员非友元函数应当是默认做法，我们应当努力做到这一点。

学会考虑上面这些（以及相关的）问题很重要，因为我们想要尽可能地将函数归到最后一种情况。

关于效率：对于每个函数，我们会考虑将它实现为非成员非友元函数是否与实现为成员函数一样高效。这算是过早优化吗（我常常批评过早优化）？一点都不是。考虑效率的主要原因是为了说明 basic_string 中的多少成员函数在改写为非成员非友元形式之后也照样高效。我特别要消除掉那些所谓“将操作实现为非成员非友元函数会有潜在的效率损失，例如使某些操作无法在标准要求的常数时间内完成”等之类的谣言！另一个原因是要说明评断一个通用意图的库的优化是否过早优化与评断一个应用程序预先进行的优化是否过早优化的标准是不一样的——你可以对一个应用程序的最终系统进行真实而明确的性能测试，但通常你对你的库会被用到什么地方、被怎样使用是没什么概念的！后一种情况下，为以防万一，最好确保任何常用操作的高效性，不过前提是这么做不会涉及接口或无谓地增加复杂性。此外，通用意图的库的实现者通常会从过去的经验中提炼出真实而具体的信息（例如以前用户的报告，或者实现者自己在相应问题领域的经验），譬如哪些操作被用户们用在时间敏感的地方，因此他的优化并不是盲目的，并且不大可能是过早优化。

293

所以，不要对过早优化神经过敏，这儿我们并没有掉进这个陷阱，而是在考察 basic_string 到底有哪些成员函数可以“无损”地实现为非成员非友元形式。我知道你可能已经料到会有很多成员函数可以被转换成非成员形式，但实际的结果可能仍然会令你大吃一惊。

哪些操作必须为成员形式

1. std::string 的哪些成员函数必须是成员形式的？为什么？

首先让我们将那些必须是成员形式的操作抽取出来。在列表的开始我们看到了一些明显的函数：

- 构造函数(6)
- 析构函数
- 赋值操作符(3)
- []操作符(2)

显然，这些函数必须是成员函数。因为标准规定，构造函数、析构函数、赋值操作符或者[]操作符都不能是非成员形式！

刚才加起来一共 12 个函数，剩下还有 91 个……

哪些操作应当为成员形式

2. std::string 的哪些操作应当为成员形式的？为什么？

哪些操作需要访问类的内部数据，且如果它们不是成员函数就得通过将它们声明为友元来给它们这个权限呢？显然这些函数是有理由成为成员函数的，而且通常也应当是成员函数。std::string 中的此类成员函数如下，其中有些提供了对 string 内部状态的间接访问（例如 begin()）或改动（例如 reserve()）：

294

- begin (2)
- end (2)
- rbegin (2)
- rend (2)
- size
- max_size
- capacity
- reserve
- swap
- c_str
- data
- get_allocator

以上函数之所以应当为成员函数不仅是因为它们与 basic_string 的关系比较紧密，还有另一个原因就是它们构成了那些非成员非友元形式的函数要使用（依赖于）的公用接口。当然，可以

将它们实现为非成员的友元函数，但为什么要这么做呢？（关于为什么应当尽量编写非成员非友元形式的函数（后者转而利用成员函数来实现）还有一个原因，即接口统一性，待会儿我们会回到这个问题）。

我打算往这个列表中再添一些函数，作为 string 的基本操作：

- insert（1——三个参数的版本）
- erase（1——erase(iter, iter)版本）
- replace（2——replace(iter, iter, num, char)以及模板版本）

待会儿我们会回到 insert()、erase()和 replace()问题。尤其是对于 replace()，一定要选择其中一个最灵活最基本的版本作为成员函数。

切入正题：可能改写为非成员非友元形式的操作

3. *解释为什么 std::string 的 at()、clear()、empty()和 length()这 4 个成员函数可以在不损失泛型性或可用性、且对 std::string 的接口的其他部分没有影响的前提下改写成非成员非友元形式。*

首先我要指出的是，以下这几个函数存在着一些根本上的共通点，即它们中的每一个都可以轻易地、无损效率地变为非成员非友元函数：

295

- at (2)
- clear
- empty
- length

当然它们可以是非成员非友元形式，这很明显，没什么好疑虑的。当然，这些函数还有另一处相当根本的共通点，即标准在关于容器和序列的要求部分将它们描述为成员函数。

“对，等等！”我几乎可以听见有些脑袋被“标准化”了的家伙说道，朝着我这边，随时准备发起一番口舌之争，“没这么快吧！难道你不记得 basic_string 的设计是要符合 C++标准中对容器和序列的要求的吗？这些要求中明确写道或建议上面这些函数中有些应当为成员函数。停止误导读者吧！这些函数是成员函数，它们就是成员函数！你没法改变这个事实，只好接受它！”是的，没错，但既然是讨论，我就打算冒天下之大不韪，抛弃刚才提到的容器和序列的要求，并赶紧走为上策。

下面就是我的观点：破天荒头一遭，我的考虑与标准对容器的要求不一致。我考虑的问题是哪些函数（在不损失效率的前提下）可以被改写为非成员非友元形式，以及这么做了以后是否能够获得额外的好处。如果对于那些被标准限定为成员形式的函数，改写为非成员非友元形式能够带来好处，那为何不顺便指出标准中的说法可以改进呢？于是我们就……

就拿 empty()来说吧。可以将 empty()实现为非成员非友元函数吗？当然……标准本身要求

basic_string::empty()具有如下行为（参见[C++03, § 21.3.3/14]）：

> 返回：**size()==0**

呃……现在看来写一个非成员形式的效率无损的 empty()实在是易如反掌：

```
template<class charT, class traits, class Allocator>
bool empty(const basic_string<charT, traits, Allocator>& s) {
  return s.size() == 0;
}
```

当然，如果不是 size()这个成员函数的支持，要想实现这么一个非成员非友元形式的 empty()是不可能的。作为替代方案（而且在一些情况下性能更可靠），还可以这样来实现：

296

```
template<class charT, class traits, class Allocator>
bool empty(const basic_string<charT, traits, Allocator>& s) {
  return s.begin() == s.end ();
}
```

或者再泛化一点：

```
template<typename T>
bool empty(const T& t) {
  return t.begin() == t.end();
}
```

最后一个版本根本就不依赖于 string，不过其模板实参类型必须提供 begin()和 end()成员函数。这表明无论如何类的公用成员函数确实得提供必要且足够的功能。后面在考虑其他函数的时候也会看到这一点（对于本条余下的示例代码，我并不打算将那些非成员非友元形式的函数模板写成完全泛化的版本，而是打算写成特别针对 basic_string 的版本。不过它们皆可以以上面的方式来加以泛化，在后面的讨论中将会看到为什么这么做实际上是有充分理由的）。

注意，尽管可以将 size()写成成员形式，并基于它来实现 empty()，但没法反过来。这里提到的几种情况中存在着一组相关的函数，其中或许有不止一个可以成为成员函数，并由它们来实现那些非成员函数。那么，究竟哪些函数应当被任命为成员函数呢？我的建议是选择那些最灵活且无损效率的，让它们成为其他函数的基石。在刚才的例子中，选择 size()作为成员函数，因为它的结果总是可以被缓存起来（实际上，标准鼓励将 size()的返回值暂存起来，因为 size()“应当”在常数时间内返回），这样，一个基于 size()来实现的 empty()与以访问 string 的内部数据结构为前提实现出来的 empty()相比在效率上没有丝毫的逊色[1]。

作为另一个例子，我们来看看 at()。同样的道理也适用于 at()。标准对 at()的 const 和非 const 版本的要求如下：

> 抛出：***out_of_range if pos >= size()***

1. 有趣的是，同样的性能分析对 list 却并不适用，因为 list::size()是在线性时间内完成的，而不是常数时间（倘若 empty()能够访问 list 的内部数据结构，就能够在常数时间内完成）。——译者注

返回：*operator[](pos)*

所以 at()很容易就可以改写为非成员函数，每个函数模板（const 和非 const 版）只需两行即可，虽说有点冗长（因为其中有许多模板参数和内嵌类型定义，还得记住 typename！）。下面就是代码：

297

```
template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::const_reference
at(const basic_string<charT, traits, Allocator>& s,
    typename basic_string<charT, traits, Allocator>::size_type pos)
{
  if(pos >= s.size()) throw out_of_range("don't do that");
  return s[pos];
}

template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::reference
at(basic_string<charT, traits, Allocator>& s,
    typename basic_string<charT, traits, Allocator>::size_type pos)
{
  if(pos >= s.size())
    throw out_of_range("I said, don't do that");
  return s[pos];
}
```

clear()呢？小菜一碟，与 erase(begin(),end())一样。我把这件工作留给你。

length()呢？照样简单，length()与 size()给出的结果一样。还有什么要注意的吗？唔……其他容器没有 length()函数，它是 basic_string 的接口所独有的，但一旦将它作为非成员函数，就可以一致地求出任何容器的 length 了。这在所讨论的特定情况下不是太有用，因为它只不过是 size()的一个同义词而已，我承认，不过值得注意的是它展示了将一个算法写成非成员形式所带来的好处，即令它们的适用范围更广。

作为小结，考虑一下将类似 at()和 empty()的函数写成成员函数与写成非成员函数之间的利害关系。首先是我们为何要这么做？难道仅仅是因为可以将这些成员函数在无损效率的前提下写成非成员非友元形式？这么做的实际的或潜在的好处有哪些？下面就是其中的一些：

（1）简单性。将它们写成非成员函数允许我们编写和维护更少的代码。我们可以只编写 empty()一次就能够一劳永逸。为什么非要重复编写 basic_string::empty()、vector::empty、list::empty()等等？甚至对每个新的符合 STL 要求的容器（可能来自第三方库、内部库甚至 C++下一代标准）也得一遍遍地进行重复的编码。使用非成员非友元形式的函数，一次写就，永不操心。

注意，以上优点也有其局限性。一方面，有些函数，如 at()，是无法给出单一复杂度的保证的，因为不同容器的 at()复杂度本就不同：map 的 at()复杂度是对数时间的（log(N)），vector 的

298

at()复杂度则是常数时间的。另一方面，或许并非所有现有或将来出现的容器都会支持泛化的自由函数所期望它们具有的接口。正如刚才所示，empty()的简单的非成员实现对于那些不提供 size()成员函数的容器无法工作[1]，这类容器或许提供了足够的功能，但接口的形式却有所不同（例如函数名字不同），遇到这样的容器，只有对 empty()进行相应的特化或重载。

（2）一致性。避免了不同容器的成员算法之间、以及一个算法的成员与非成员版本之间毫无理由的不一致性（[Meyers01]中指出了目前存在的一些函数的成员与非成员版本之间的不一致性）。如果需要可定制行为，对非成员模板进行特化或重载应当能够达到目的。

（3）封装性。改善了封装性（[Meyers00]中对此进行了强有力的、详细的论证）。

话虽如此，难道这种做法就没有什么潜在的缺点吗？确实有两点（如下），不过在我看来它们与上面列出的优点相比显得微不足道：

第一个潜在的缺点是名字空间污染。由于 empty()是个如此常见的名字，因此将它放在名字空间作用域内有污染名字空间的危险。毕竟，难道每个名为 empty()的函数都希望具有与我们的 empty()一样的语义吗？这种说法某种程度上没错，但在两个方面显得不够强有力：首先，鼓励同名函数具有一致的语义是件好事情；其次，重载在解决二义性问题方面很擅长，名字空间污染从来也没有像过去有些人所认为的那样是个大问题。真的，把所有的名字放在同一个地方（名字空间）并共享实现的做法并非是污染该名字空间，而是对这些函数进行了“净化”，因为将它们集中在一起有助于避免 Meyers 所提到的没必要、没理由的不一致性。

第二是一致性问题。你可能会说将类似 empty()的函数实现为成员函数是遵循了“最小意外”原则，因为类似的函数也是成员函数，难道要写惯了 str.size()的人每次都记住写 length(str)吗？这一论点看上去似乎挺有说服力，但听听 Meyers 的建议吧：“只要是可能且合理的，就要将函数编写成非成员形式，即便你不得不同时编写相应的成员版本”，所以如果人们原本就遵循了 Meyers 的建议，这根本就不会带来什么不一致性。 299

特别地，假设为那些本是成员形式的函数添加非成员形式的版本，如 size()，在实现它们的非成员版本时只需简单地转发至相应的成员版本即可，这么一来就统一了调用语法，并解决了上面所说的一致性问题。例如，如果额外提供了一个非成员版本的 size()，那么 size(str)和 str.size()就都是可用的，这样就不再需要记住哪些函数是成员形式的了。于是人们就会慢慢习惯于非成员调用语法，并享受其简单性、一致性以及封装性优势所带来的好处。

顺便一提，除了上面提到的原因之外，还有其他技术和设计上的考虑是倾向于使用非成员形式的。一方面，默认且一致地提供非成员函数能够显著地简化模板的编写。如果模板代码中调用到的函数对于所有类型来说都是非成员形式，而不是对于某些类型是成员形式而对另一些类型是非成员形式，在编写该模板时就可以避免使用某些类似 traits 的技巧来判断对某个类型使用成员

1. 当然，这种容器不符合标准对 STL 容器的要求，但可能有些旧式容器（或新式容器）虽然不是十分 STL，但也有它们自身的用场，故而或许我们也想支持这类容器。

形式还是非成员形式的函数了（参见 *More Exceptional* C++ [Sutter02]的第 37 条中更为详细的例子）。另一方面，它会令一个操作的成员版本实现和非成员版本实现之间发生重载——如果你听到这里又有点“毛骨悚然”，准备大力抨击其不是，我只能告诉你，经验表明这一点常常会带来好处，标准委员会的一些成员觉得这种成员和非成员之间的重载应当被加入 C++0x（这不一定会发生，我只不过是告诉你就连某些专家都觉得这可能是个好主意）。

因此，上面所谓的“影响调用形式的一致性”，其实归结于一个问题：我们是应当牺牲一些实际的好处来顺从成员命名的传统呢，还是改变这个传统并享受实实在在的好处？退一万步说，至少这一说法是缺乏说服力的，因为一旦将 Meyers 的建议作为一般法则来遵守，就只会获得更好的一致性。

在解决了这些颇具争议的问题之后，接下来的一条将会继续讨论其他可以写成非成员非友元形式的函数。其中有些也是被标准要求为成员形式的，但这里我们并不是在考虑标准要求必须做到的，而是假设给我们的是一张白纸，我们自己来选择是将这些函数设计为成员形式还是非成员非友元形式。

300

第 39 条　分解 std::string 之三：给 std::string 瘦身　难度：5

我们看到，std::string正在迅速地“苗条”起来。

JG问题

1. string::resize()可以为非成员函数吗？请解释。

Guru问题

2. 分析 std::string 的下列成员函数，并说明它们是否应当改写为非成员函数。说出你的理由。

（a）assign 和 +=/append/push_back

（b）insert

解决方案

更多可成为非成员非友元形式的操作

从本条将会看到，以下所有的函数都可以实现为非成员非友元形式：

- resize (2)
- assign (6)
- += (3)
- append (6)
- push_back
- insert (7——除了三个参数的版本之外，其他都是)

下面就来详细考察一下：

resize

1. string::resize()可以为非成员函数吗？请解释。 301

唔……让我们来看看：

```
void resize(size_type n, charT c);
void resize(size_type n);
```

这两个 resize()都可以写成非成员非友元形式吗？当然可以，因为它们都可以基于 basic_string 的公用接口来实现，且无损于效率。实际上，标准本身在规定这两个版本的 resize()

的行为时也是这么做的。例如：

```
template<class charT, class traits, class Allocator>
void resize(basic_string<charT, traits, Allocator>& s,
            typename Allocator::size_type n, charT c)
{
  if(n > s.max_size()) throw length_error("won't fit");
  if(n <= s.size()) {
    basic_string<charT, traits, Allocator> temp(s, 0, n);
    s.swap(temp);
  } else {
    s.append(n - s.size(), c);
  }
}

template<class charT, class traits, class Allocator>
void resize(basic_string<charT, traits, Allocator>& s,
            typename Allocator::size_type n)
{
  resize(s, n, charT());
}
```

这是将它们“无损”实现为非成员非友元函数的途径之一。

assign和+=/append/push_back

2. 分析 std::string 的下列成员函数，并说明它们是否应当改写为非成员函数。说出你的理由。

（a）assign和+=/append/push_back

首先来看一下 assign()：我们一共有……唔……6 个版本的 assign()。所幸情况还算简单：它们本身互相之间就有着依赖关系[C++03;21.3.5.3]，而且它们全都可以由一个构造函数和 operator=()来实现。

302

接下来，operator+=、append()和 push_back：这些不同“口味”的 append 操作该怎么办呢？它们分别是：operator+=()3 个、append()6 个以及 push_back()1 个。只要看它们之间的这种相似程度，我们大概就能猜到或许其中只有一个需要被实现为成员函数。毕竟，它们本质上做的是同样的事情，虽说细节上稍有不同。例如，添加一个字符是一种情况，添加一个 string 又是一种情况，而添加一个迭代区间仍然是一种情况。实际上，它们皆可以被类似地无损实现为非成员非友元形式：

- 显然 operator+=()可以基于 append()来实现，因为 C++标准中正是这么说的。
- 同样明显，append()的 6 个版本中的 5 个都可以被改写为非成员非友元版本，因为标准指出用三参的 append()版本来实现它们，而三参的 append()版本进一步又可以由 insert()来实现，这就将 append()家族成员尽数推上了非成员化的道路。

- push_back()则需要多费点力气，因为标准在 basic_string 这一节并没有指出该操作符的语义，而是在容器要求部分指出的，从后者中发现 a.push_back(x)其实只不过是 a.insert(a.end(),x)的同义词。

“等等！”有些人可能会说：“C++标准说赋值操作符必须为成员形式的，而+=正是个赋值操作符啊！”对，但也不对。大致上说，虽然在 C++标准中+=与所有其他*=操作符是罗列在一块统称为赋值操作符的，但实际上只有 operator=()必须是成员形式的重载。因此，下列简单的非成员形式的 operator+=()重载是完全合法的 C++代码：

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s,
           const basic_string<charT, traits, Allocator>& t) {
 return s.append(t);
}

template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s, const charT* p) {
 return s.append(p);
}

template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s, charT c) {
 return s.append(1, c);
}
```

303

那么，到底是什么使得我们可以将所有这些函数统统都实现为非成员非友元形式的呢？答案是 insert()。所以，将 insert()作为成员函数是良好的选择，因为它既做了最基本的工作，又将所有与“添加”有关的对 string 内部结构的访问全都集中在这一个点上，而不是分散在各个不同的函数当中。

insert

（b）insert

下一个是 insert：如果你觉得 assign()共有 6 个版本、append()也有 6 个版本似乎有点嫌多，我只能说与 insert()比起来它们又都是小儿科了，因为 insert()一共有 8 个版本。

我已经将三参的 insert()设定为成员函数，现在来说说为什么要这么做。首先，正如前面提到的，insert()比 append()更具一般性，更通用。只需将一个 insert()设为成员函数就可以将所有 append 操作设为非成员非友元形式；其次，设想如果每个 insert()都不是成员形式，那么免不了还是得从 append()重载版本中至少选一个出来担当此任，所以我最终选择将更为基本也更灵活的操作（即三参版本的 insert()）作为成员函数。

但问题是我们有 8 种不同“口味”的 insert()，到底该选哪个（或哪些）作为成员版本呢？标准已经为其中的 5 个重载版本指明了命运：它们都是基于三参版本的 insert()来实现的。同时，考虑到其他的 insert()也都可以高效地由三参版本的 insert()来实现，因此只需将后者实现为成员函数，其他皆可以实现为非成员非友元函数。

中场休息

你可能觉得像 insert()这样多达 8 个重载版本的操作已经是极限了，但我要告诉你，这也只不过是热身。待会儿我们将会看到 replace()洋洋洒洒的 10 个版本。不过，在着手之前，先休息一会儿。下一条首先会从一个较为简单的函数（erase()）开始，因为 erase()的解决对于后面顺利处理 replace()是有帮助的……

304

第 40 条　分解 std::string 之四：再论 std::string　难度：6

本条款我们完成最后一部分的讨论，并最终将一个“精简版”的std::string呈现在你的面前。

JG问题

1. string::erase()可以写成非成员函数吗？请解释。

Guru问题

2. 分析 std::string 剩下的成员函数（如下），并说明它们是否应当改写为非成员形式。说出你的理由。

（a）replace

（b）copy 和 substr

（c）compare

（d）<find>(find, find_* 和 rfind)[1]

解决方案

更多可以实现为非成员非友元版本的操作

本条款我们完成最后一部分讨论，终点就在眼前。只剩下很少几个函数还没有解决，我们最终会看到，它们全都可以被实现为非成员非友元形式。

- erase (2——除了 erase(iter,iter)之外的其他所有重载版本)
- replace (8——除了 replace(iter, iter, num, char)以及模板版本之外的其他所有重载版本)
- copy
- substr
- compare (5)
- find (4)
- rfind (4)
- find_first_of (4)

305

1. <find>表示与 find 相关的一族函数，如 find/find_if/rfind 等。——译者注

- first_last_of (4)
- find_first_not_of (4)
- find_last_not_of (4)

咖啡时间："抹掉" erase()

1. string::erase()可以写成非成员函数吗？请解释。

上一条款的最后我曾提到本条款会先对付 erase()：在讨论了一共约 30 种不同形式的 assign()、append()、insert()以及 replace()，并对付了其中 20 个之后，得知 erase()一共才 3 个版本，而且只有其中两个是属于本节讨论的范畴之后，你大概有种如释重负的感觉吧。比起上一条款费了九牛二虎之力，这里我们就算理当停下来享受一杯咖啡吧……

erase()的情况有点意思：其中至少有一个版本应当充当成员函数（或友元）的角色，因为使用前面提到的任何成员函数都无法高效地实现 erase()。实际上 erase()函数也分两个派系：

```
// erase(pos, length)
basic_string& erase(size_type pos = 0, size_type n = npos);

// erase(iter, ...)
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

首先，注意到这两派的 erase()之间在返回值类型上是不一致的：第一个 erase()返回 string 的引用；而另外两个则返回一个迭代器，它指向被 erase 掉的区间或字符的后一个元素。其次，注意这两派 erase()的实参类型也是不一样的：第一个 erase()接受的两个参数分别表示偏移量（erase 的起点）和长度（被 erase 的区间长度）；而其他两个 erase()则接受一个迭代器或迭代器区间为参数；幸运的是，我们可以通过 pos = iter-begin()来将迭代器转换为偏移量，也可以通过 iter = begin() + pos 来将偏移量转换为迭代器。

（题外话：标准并不要求 basic_string 将其数据存放在内存中连续的 charT 数组缓冲区中，但特定的实现可以选择这么做。如果这么做了，那么从迭代器转换为正偏移（或反之）将会是没有任何额外开销的。不过，就这个问题而言，我敢说，即便 string 内部采用的是分段式存储策略，也可以通过容器和迭代器的公用接口提供偏移量跟迭代器之间的高效转换。不过这跟我们所讨论的"成员和非成员非友元"问题就没太大关系了，我这里提到它纯粹是为了讨论的完整性。）

306

照这种说法，erase()的前两个版本就可以基于第三个版本来实现（再次提醒别忘了那些 typename 和名字限定！），如下：

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
erase(basic_string<charT, traits, Allocator>& s,
        typename Allocator::size_type pos = 0,
        typename Allocator::size_type n =
```

```
        basic_string<charT, traits, Allocator>::npos)
{
 if(pos > s.size())
    throw out_of_range("yes, we have no bananas");
 typename basic_string<charT, traits, Allocator>::iterator
    first = s.begin()+pos,
    last = n == basic_string<charT, traits, Allocator>::npos
            ? s.end() : first + min( n, s.size() - pos );
 if( first != last )
    s.erase( first, last );
 return s;
}

template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::iterator
erase(basic_string<charT, traits, Allocator>& s,
        typename basic_string<charT, traits, Allocator>::iterator position)
{
 return s.erase( position, position+1 );
}
```

OK，咖啡时间结束……

回到工作：替换replace()

2. 分析 std::string 剩下的成员函数（如下），并说明它们是否应当改写为非成员形式。说出你的理由。

（a）replace

下一个，replace()：说实话，这个多达 10 个版本的 replace()函数族比较无趣，说是冗长乏味也不为过。

在这些形形色色的 replace()当中，至少有三个必须得是成员（或友元）形式的，不然使用前面提到过的其他函数是没法高效地实现 replace()的。有一点尤其值得注意，即你并不能使用 erase()加上 insert()来高效地实现 replace()，反之亦然，因为跟成员函数实现相比，这两种做法都需要涉及更多的字符（元素）重组，甚至可能会导致缓冲区重分配。 307

replace()函数也分为两大派系：

```
// replace(pos, length, ...)
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);          // #1
basic_string& replace(size_type pos1, size_type n1,                                    // #2
                      const basic_string& str, size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2);      // #3
basic_string& replace(size_type pos, size_type n1, const charT* s);                    // #4
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);             // #5
```

```
// replace(iter, iter, ...)
basic_string& replace(iterator i1, iterator i2, const basic_string& str);          // 6
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);      // 7
basic_string& replace(iterator i1, iterator i2, const charT* s);                   // 8
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);             // 9
template<class InputIterator>                                                      // 10
  basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);
```

跟 erase()不同的是，replace()的两大派系的返回类型是一致的；这让我们多少感觉轻松了一点。不过，跟 erase()一样，这两派的 replace()的参数类型也是不统一的：第一派是基于“偏移量+长度”来实现的，而另一派则是基于迭代器区间。不过由于我们可以在迭代器和偏移量之间进行转换，所以我们可以轻而易举地用其中一派的函数来实现另一派的。

在考虑哪些函数必须为成员时，我们的意愿是选择最为灵活也最基本的版本作为成员函数，并基于它们来实现其他的非成员非友元形式的版本。不过这一分析方式也存在一些小缺陷，下面就为你展示它们，并说明如何避免。譬如首先我们考虑 replace()的第一组重载，看看应当选用哪几个或一个来作为成员版本：

- 只选其中一个（#2)？有人可能注意到标准中是基于#2 来制定其他几个版本的语义的。然而遗憾的是，如果真的这么实现的话，有些函数在将工作转发给#2 的时候会免不了创建 basic_string 的临时对象，因此我们不能光选#2（即便只是对第一组重载而言）。标准所指出的只是可观察的行为，而并非实际实现一个给定函数的最佳途径。
- 选其中的两个函数（#3 和#5）？有人可能注意到，在第一组当中除了#5 之外其他版本都可以高效地基于#3 来实现，不过#5 必须得独立出来，因为如果也用#3 来实现它的话，就会同样遇到临时对象的问题（见上一条）。

接着考虑第二组重载：

308

- 只选其中一个（#6）？你可能注意到标准是基于#6 来制定其他几个版本的语义的。然而遗憾的是，这同样面临前面所说的临时对象问题，因此我们也不能光选#6（即便只是对第二组重载而言）。
- 选其中的三个函数（#7、#9、#10）？你可能注意到第二组中的大多数函数，除了#9 和#10 之外，都可以高效地通过#7 来实现。其中#9 不能的原因跟前面所说的#5 的原因一样，即如果#9 要用#7 来实现的话，调用#7 的时候必须给它一个缓冲区，而实际上用户在调用#9 的时候是不会给出现有的缓冲区的，这就导致#9 得自己创建一个临时的缓冲区；#10 不能用#7 来实现的原因则是它根本不能假定它收到的迭代器是简单的指针或者甚至是 basic_string::iterator[1]。
- 选其中的两个函数（#9 和#10）！你可能立即会注意到第二组中除了#9 之外的所有函数都可以由#10 来实现，包括#7。实际上，如果假定 basic_string 的底层内存布局是连续的，

1. 请参看上文代码中#10 的声明。——译者注

并且偏移量和迭代器之间也能够高效地进行相互转换的话，那么我们甚至可以用#9 跟#10 来实现第一组中的所有重载版本。所以……啊哈！就选它们了！

那么，看起来我们最多能做到的就是选出其中两个函数作为成员函数，并让其他所有 8 个非成员非友元函数基于它们来实现。被选为成员的两个函数分别是 replace(iter,iter,num,char)以及那个惟一的模板版本。其他剩下的所有版本都写成非成员非友元形式（我把为其他 8 个版本编写简单高效的非成员实现的任务留给读者）。

注意，#10 优雅地展示了模板的强大威力，我们可以使用它来无损效率地实现其他 7 个重载版本，而且即使用它来实现剩下的两个版本（#5 和#9）损失的效率也可忽略不计（构造一个临时的 basic_string，其中包含了 n 个同样的字符）。

又到了我们的咖啡时间了……

咖啡时间#2：复制copy和substr

（b）copy 和 substr

哦，copy()，讨厌的家伙。copy()有点像是一个反常的家伙，它的接口跟 std::copy()算法的接口是不一致的。我们注意看它的函数类型：

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

上面这个 copy()是 const 成员函数，它并不改变 string 本身。其作用是从 string 对象身上取一段出来（最多 n 字符，从 pos 所代表的偏移处开始）复制到 s（目标缓冲区）当中去（注意，我故意没说“C 风格串”，而是说“目标缓冲区”），后者要足够大，否则程序就会篡改跟在缓冲区末尾之后的内存，结果是未定义行为。而且更糟的是，basic_string::copy()在往目标缓冲区中复制完了字符之后并不会加上一个结束符，这就是我刚才不称它为“C 风格串”的原因（另一个原因是 charT 并不一定得是 char；这个函数可以往缓冲区中复制任何构成该 string 对象本身的字符类型变量）。不加结束符也是导致 copy()成为一个危险的函数的根由之一。 309

准则：永远不要使用那些在写缓冲区时不进行越界检查的函数（例如strcpy、sprintf等）或那些可能不会为C风格串加上结束符的函数（例如strncpy、basic_string::copy等）。它们非但是潜伏的危胁，更是明显而迫在眉睫的安全灾难——缓冲区溢出攻击永远都是黑客和恶意代码编写者钟情的武器。[第2条和第3条有更多关于该主题的讨论]

实际上，我们只需使用 std::copy()，所有必要的工作都可以以几乎同样简单的方式来完成，而且要灵活得多：

```
string s = "0123456789";
char* buf1 = new char[5];
s.copy(buf1, 0, 5);                    // ok: buf将会包含'0', '1', '2', '3', '4'
copy(s.begin(), s.begin()+5, buf1);
```

```
                                        // ok: buf将会包含'0', '1', '2', '3', '4'
int* buf2 = new int[5];
s.copy(buf2, 0, 5);                     // 错误，第一个参数的类型并非char*
copy(s.begin(), s.begin()+5, buf2);
                                        // ok: buf2将会包含以下字符的ASCII值:
                                        //     '0', '1', '2', '3', '4'
```

以上代码也顺带说明了将 basic_string::copy()改写为非成员非友元形式有多简单，如果借助于 std::copy()来实现它的话则最为简单。（我把这个留给读者作为练习，不过别忘了正确处理 n == npos 这种特殊情况）。

好，basic_string 的这个成员函数倒没给我们伤什么脑筋，所以趁着还轻松，让我们接着来看另一个简单的成员函数：substr()。我们回顾一下它的函数原型[1]：

310

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

注意，substr()可以轻易改写为非成员非友元形式，因为它只不过是 basic_string 构造函数的一层语法糖衣而已。毕竟，标准本身也说了，substr()必须返回一个新生的 basic_string 对象，它由 basic_string<chart ,traits ,Allocator>(data()+pos ,min(n,size()-pos))的等价形式来构造。这也就是说，创建一个新的串对象并使其内容等于已有串对象的子串，这一功能也可以不经过 substr()来实现：只需使用更为一般化的 basic_string 构造函数就可以实现，后者本就具有这样的功能，而且还有其他的一些灵活性，如下：

```
string s = "0123456789";

string s2 = s.substr( 0, 5 );           // s2包含"01234"
string s3( s.data(), 5 );               // s3包含"01234"
string s4( s.begin(), s.begin()+5 );    // s4包含"01234"
```

好吧，休息结束。回到工作上来……令人开心的是现在只剩下两族函数需要对付了，它们就是 compare()和 find*()。

接近尾声：比较compare()

（c）compare

我们要考虑的倒数第二组成员函数是 compare()。它由五个重载版本组成，其中每个都可以高效地实现为非成员非友元形式。然而究竟怎么实现呢？考虑到在标准当中它们是基于 basic_string 的 size()和 data()来制定的，而这两者我们已经决定做成成员函数，另外还有 traits::compare()，实际的工作就是由它完成的（文献[Josuttis99]中有更多关于 traits::compare()

1. 敏锐的读者可能已经注意到了，该函数是按照“位置，长度”这一顺序来接受参数的，而刚才我们看到的 copy()也是接受的同样的参数，但顺序刚好相反，是“长度，位置”。除了在外观上不一致之外，这实际上也可能会带来危险性。由于 basic_string 的用户需要记住哪个函数是按什么顺序来接受参数的，所以容易误入陷阱。又因为这两个参数碰巧类型相同，所以可怜的用户就算搞反了也会顺利编译并运行其程序……只不过运行着运行着就会冒出奇怪的行为，例如导致选取了错误的子串，进而输出了错误的串等。对此心怀不满的人可能会觉得，这就好比将一个贴着“由标准委员会设计”标签的地雷埋在路边，不加警告，就等着人们去一脚踏上……呀！抱歉，离题了。

的描述）。

唉！等一下。这听起来倒挺简单的！现在请不要提出什么疑问，我们先接着往下看：

终点：寻找find()

（d）*<find>*（find,find_*和rfind）

唉！我们的轻松并没有持续多久。惟恐前面遭的罪还不够似的，这下我们看到的是一个包含24个成员的大家族（我没骗你）：

find 函数共分 6 族，每族包含 4 个成员：

- find()。从 pos 点开始正向搜索某个给定串或字符(str、s 或 c)首次出现的地点。
- rfind()。从 pos 点开始反向搜索某个给定串或字符(str、s 或 c)首次出现的地点。 311
- find_first_of()。从 pos 点开始正向搜索某个给定串中任一字符或给定字符(str、s 或 c)首次出现的地点。
- find_last_of()。从 pos 点开始反向搜索某个给定串中任一字符或给定字符(str、s 或 c)首次出现的地点。
- find_first_not_of()。从 pos 点开始正向搜索不在给定串所包含的字符集合中或不等于给定字符的任何字符（并返回其所在位置）。
- find_last_not_of()。从 pos 点开始反向搜索不在给定串所包含的字符集合中或不等于给定字符的任何字符（并返回其所在位置）。

每族包含 4 个成员，调用形式分别如下：

- <find>(str,pos)。其中 str 是搜索的目标串，pos 是搜索的起点。
- <find>(ptr,pos,n)。其中 ptr 是 charT*类型的指针，指向一个长为 n 的缓冲区，后者包含有待搜索的字符，pos 为搜索起点。
- <find>(ptr,pos)。其中 ptr 是 charT*类型的指针，指向一个 null 结尾的缓冲区，后者包含有待搜索的字符，pos 为搜索起点。
- <find>(c,pos)。其中 c 为有待搜索的字符，pos 为搜索起点。

所有这些函数都可以高效地实现为非成员非友元形式：我把具体实现的任务留给你作为练习。然后还有什么呢？没有了！就这么多！

不过请允许我对 basic_string 的 find 最后再说两句：实际上，你可能已经注意到了，除了basic_string::<find>算法族之外，标准其实还提供了虽说不像它们那样全面详尽但也比较丰富的std::<find>算法族。特别地，它大致包含如下版本：

- std::find()可以完成 basic_string::find()的工作。
- std::find()+reverse_iterator 或 std::find_end()可以完成 basic_string::rfind()的工作。
- std::find_first_of()或 std::find 辅以合适的谓词（predicate）就可以完成 basic_string::find_first_of()的工作。

- std::find_first_of()或 std::find()加上合适的谓词，辅以 reverse_iterator，就可以完成 basic_string::find_last_of()的工作。
- std::find()辅以合适的谓词可以完成 basic_string::find_first_not_of()的工作。
- std::find()辅以合适的谓词跟 reverse_iterator 可以完成 basic_string::find_last_not_of()的工作。

312

还有什么呢？非成员形式的算法更为灵活，因为它们不仅对 basic_string 有效。实际上，所有的 basic_string::<find>算法都可以通过 std::find()、std::find_end()辅以合适的谓词以及（如果必要的话）reverse_iterator 来实现。

那么，为何不直接甩掉 basic_string::<find>这个包袱，直接叫程序员去使用现有的 std::<find>算法族呢？当然，但有些地方你得注意：其一，虽说 basic_string::<find>的工作可以由 std::<find>辅以适当的设施来完成，但有些情况下使用 std::<find>的默认实现来做可能会导致显著的性能损失。find()和 rfind()的三种搜索子串（不仅仅是搜索单个字符）的版本如果不用每次迭代一步的蛮力搜索法来实现的话，效率可以高得多。这方面有些非常有名的算法就是通过先即时生成有限状态机，然后将其应用到源串上来进行搜索，从而获得线性时间复杂度。利用这些算法来改进 find 或许是有必要的。

为了利用起上述的算法优化，我们可不可以提供专为 basic_string::iterator 准备的重载版本（不是特化，见第 7 条）呢？可以，但只有当 basic_string::iterator 是类类型而不是简单的 charT*的时候才可以。原因是，如果它是个简单的指针类型的话，我们为它特化 std::find()就等于是针对所有该类型的指针特化了 std::find()，这显然是错误的，因为并非所有字符指针都一定指向 std::string 内部。因此要想侦测 basic_string::iterator 并对其进行特化的话，basic_string::iterator 就必须得是一个单独的自定义类型。这样我们的特化版才能够放心地进行算法优化，并在子串匹配上达到完全的高效。

小结

解耦和封装都是好事。特别地，将算法跟容器分离开来通常是有益的做法，STL 大多数时候正是这么做的。

人们广泛接受的一个事实是，basic_string 具有太多的成员函数了。在它的共计 103 个成员函数之中，只有 32 个真的必须是成员函数，剩下 71 个皆可以无损效率地改写为非成员非友元形式。实际上，这些成员函数中有许多都是在无谓地重复 std 算法已经提供了的功能，或者其本身就可以从 basic_string 上脱离出来（而不是牢牢地捆绑在上面），成为更有用、适用范围更广的算法。

313

不要在你自己的设计中重复犯 basic_string 所犯的错误。从容器中将算法解耦出来，如果有正当理由，可使用模板特化或重载来定制特殊意图的行为（像上面提到的子串查找那样），不过首先要遵循下面这些原则——用户会感激你的，因为你为他们省了很多工夫。

准则： **尽可能遵守单一职守原则。**

314 **尽可能编写非成员非友元形式的函数。**

参 考 文 献

注：为浏览方便，本参考文献也可网上查阅，网址为：

http://www.gotw.ca/publications/xc++s/bibliography.htm

粗体显示的参考文献（如**[Alexandrescu02]**）在上述网址中提供了超链接。

[Alexandrescu01] A. Alexandrescu. *Modern C++ Design* (Addison-Wesley, 2001).

[Alexandrescu02] A. Alexandrescu. "Discriminated Unions (I)," "... (II)," and "... (III)" (*C/C++ Users Journal*, 20(4,6,8), April/June/August 2002).

[Arnold00] M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney. "Adaptive Optimization in the Jalapeño JVM" (*Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications,* ACM Press, 2000).

[Bentley00] J. Bentley. *Programming Pearls, Second Edition* (Addison-Wesley, 2000).

[Boost] C++ Boost (www.boost.org).

[BoostES] "Boost Library Requirements and Guidelines, Exception-specification rationale" (Boost web site).

[Cargill94] T. Cargill. "Exception Handling: A False Sense of Security" (*C++ Report,* 9(6), November-December 1994).

[C90] ISO/IEC 9899:1990(E), *Programming Languages*—C (ISO C90 and ANSI C89 standard).

[C99] ISO/IEC 9899:1999(E), *Programming Languages*—C (revised ISO and ANSI C99 standard).

[C++98] ISO/IEC 14882:1998(E), *Programming Languages*—C++ (ISO and ANSI C++ standard).

[C++03] ISO/IEC 14882:2003(E), *Programming Languages*—C++ (up-dated ISO and ANSI C++ standard including the contents of [C++98] plus errata corrections).

[C++CLI04] *C++/CLI Language Specification, Working Draft 1.6* (Ecma International, August 2004).

[Cline99] M. Cline, G. Lomow, and M. Girou. *C++ FAQs, Second Edition* (Addison-Wesley, 1999).

[Coplien92] J. Coplien. *Advanced C++ Programming Styles and Idioms* (Addison-Wesley, 1992).

[Dewhurst02] S. Dewhurst. "C++ Hierarchy Design Idioms" (*Software Development 2002 West conference talk,* April 2002).

[Dewhurst03] S. Dewhurst. *C++ Gotchas* (Addison-Wesley, 2003).

[Ellis90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual* (Addison-Wesley, 1990).

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

[GotW] H. Sutter. *Guru of the Week*

[Hicks00] C. Hicks. "Creating an Index Table in STL" (*C/C++ Users Journal,* 18(8), August 2000).

[Hyslop00] H. Sutter and J. Hyslop. "Virtually Yours" (*C/C++ Users Journal,* 18(12), December 2000).

[Hyslop01] H. Sutter and J. Hyslop. "I'd Hold Anything For You" (*C/C++ Users Journal,* 19(12), December 2001).

[JikesRVM] Jikes RVM home page.

[Jones96] R. Jones and R. Lins. *Garbage Collection* (Wiley, 1996).

[Josuttis99] N. Josuttis. *The C++ standard Library* (Addison-Wesley, 1999).

[Kalev01] D. Kalev. "Designing a Generic Callback Dispatcher" (DevX, 2001).

[Koenig96] A. Koenig, "When Memory Runs Low" (*C++ Report,* 8(6), June 1996).

[Langer00] A. Langer and K. Kreft. *Standard C++ IOStreams and Locales* (Addison-Wesley, 2000).

[Lippman98] S. Lippman and J. Lajoie. *C++ Primer, Third Edition* (Addison-Wesley, 1998).

[Liskov88] B. Liskov. "Data Abstraction and Hierarchy" (*SIGPLAN Notices,* 23(5), May 1988).

[Manley02] K. Manley. "Using Constructed Types in Unions" (*C/C++ Users Journal,* 20(8), August 2002).

[Martin95] R. C. Martin. *Designing Object-Oriented Applications Using the Booch Method* (Prentice-Hall, 1995).

[Marrie00] L. Marrie. "Alternating Skip Lists" (*Dr. Dobb's Journal,* 25(8), August 2000).

[Meyers96] S. Meyers. *More Effective C++* (Addison-Wesley, 1996).

[Meyers97] S. Meyers. *Effective C++, Second Edition* (Addison-Wesley, 1997).

[Meyers99] S. Meyers. *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs* (Addison-Wesley, 1999).

[Meyers00] S. Meyers. "How Non-Member Functions Improve Encapsulation" (*C/C++ Users Journal,* 18(2), February 2000).

[Meyers01] S. Meyers. *Effective STL* (Addison-Wesley, 2001).

[Newkirk97] J. Newkirk. "Private Interface" (Object Mentor, 1997).

[ObjectMentor] Object Mentor Inc.

[Stroustrup88] B. Stroustrup. "Parameterized Types for C++" (*Proc. USENIX Conference,* Denver, October 1988).

[Stroustrup94] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[Stroustrup99] B. Stroustrup. "Learning Standard C++ as a New Language" (*C/C++ Users Journal,* 17(5), May 1999).

[Stroustrup00] B. Stroustrup. *The C++ Programming Language, Special Edition* (Addison-Wesley, 2000).

[Sutter99] H. Sutter. "ACID Programming" (Guru of the Week #61, September 1999).

[Sutter00] H. Sutter. *Exceptional C++* (Addison-Wesley, 2000).

[Sutter02] H. Sutter. *More Exceptional C++* (Addison-Wesley, 2002).

[Sutter02a] H. Sutter. "The Group of Seven: Extensions Under Consideration for the C++ Standard Library" (*C/C++ Users Journal Experts Forum*, 20(4), April 2002).

[Sutter02b] H. Sutter. "Smart(er) Pointers" (*C/C++ Users Journal*, 20(8), August 2002).

[Sutter02c] H. Sutter. "Standard C++ Meets Managed C++" (*C/C++ Users Journal, C++ .NET Solutions Supplement*, 20(9), September 2002).

[Vandevoorde03] D. Vandevoorde and N. Josuttis. *C++ Templates: The Complete Guide* (Addison-Wesley, 2003).

索　引

索引中的页码为英文原书页码，与书中边栏页码一致。

A

Abrahams Guarantees(Abrahams 异常安全保证)，85
Abrahams，David，45，85
accessibility（可访问性），111
  accessibility checking（可访问性检查），114
  vs. visibility（与可见性），113，116
ADL。参见 Koenig 查找
algorith
  spelling "algorithm"（正确的拼写是"algorithm"），249
algorithms（算法）
  copy，7
  for_each，7
  vs. member functions（与成员函数），310
Align（对齐），283
alignment（对齐），272
allocation（分配），156
  and alignment（对齐），159，272
  and deque，159，200
  and list，200
  and map，159，200
  and padding（填充），157
  and set，160，200
  and vector，159，200
  by compiler's run-time library(通过编译器的运行时库)，154
  by operating system(通过操作系统)，154
  by standard containers and allocators（通过标准容器和分配器），154
  by user-defined containers and allocators（通过用户自定义的容器和分配器），154
  combining strategies(组合策略)，153
  failure recovery（失败恢复），180
  failures，caveats about checking（失败，以及关于失败检查的告诫），176
  fixed-size（固定大小的），153
  garbage-collected（垃圾收集），153
  general-purpose（通用意图的），153
  housekeeping overhead（底层细节信息的开销），156
  lazy（迟提交），177
  value of checking failures（失败检查的意义），180
ambiguity(二义性)。见 disambiguation，见 declaration，vs. definition
Animal，230
any，281
any_cast，281
argument-dependent lookup(参数依赖查找)。见 Koenig lookup
arrays（数组）
  and alignment（和对齐），159
  and padding（和填充），157
at
  vector。见 vector，at
auto，218
auto_ptr
  prefer shared_ptr instead of（尽量使用 shared_ptr），145

B

bad_exception，94
BadDerived，147
BaitAndSwitch，107
base classes（基类）
  why not make concrete（为何不能是具体类），137
basic guarantee（基本保证），85
basic_string，286
  algorithms vs. member functions(算法与成员函数)，310
  at，writing as nonmember function(将 at()写成非成员形式)，298
  empty，writing as nonmember function（将 empty()写成非成员形式），296
  erase，writing as nonmember function（将 erase()写成非成员形式），307
  operator+=，writing as nonmember function（将 operator+=()写成非成员形式），304
  resize，writing as nonmember function（将 resize()写成非成员形式），302
binary_function，136，262
bind2nd，29
binders，28
BitBuffer，204
Boost
  any，280
  lexical_cast，22，见 lexical_cast
  shared_ptr。见 shared_ptr
Borland，53，278，283
buffer overrun（缓冲区溢出），12，310
byte（字节）
  Size of in bits(按位计算的大小)，205

C

C#，196
Calc，110
callback，260
callbackbase，264
capacity
  vector。见 vector，capacity
Cat，230
catch，81，见 exception safety
Cfront，61，70
checked_delete，49

chess（国际象棋）
example，199
circular reference（循环引用）。见 reference，circular
clarity（清晰性），208，247
class（类）
access to internal members（访问内部成员），104
class templates（类模板），42
partial specialization（偏特化），43
prefer implementing function templates using（尽量用类模板实现函数模板），47
classes（类）
accessibility of members（成员可访问性），111
code clarity（代码清晰性），208，247
Colossal Cave
allusion to（提及），32
Comeau，53，59，278，283
Compare1stDeref，255
Compare Deref，256
comparing iterators（比较迭代器）。见 iterators，comparing
compression（压缩）。见 efficiency，compression
const，6
and optimization（和优化），184，188
and ROM（和 ROM），186
avoid passing by-value parameters as（避免按值传递 const 参数），189
correctness（const 正确性），184
const correctness（const 正确性），6，184，251，261，276
const_iterator，6
construct（构造），33
constructor（构造函数）
explicit（显式），260
implicitly generated（隐式生成的），145
constructors（构造函数）
order of execution（执行顺序），100
contiguous（内存连续性）
meaning of for allocations and arrays（对分配和数组的意义），158
conversions（转换）
floating point（浮点数的），229
narrowing（窄化的），229
copy（复制），7
copy assignment（复制赋值）
implicitly generated（隐式生成的），145
suppressing（抑制），147
CoughItUp，116
Count，139，146
Couple，119
cute critters（可爱的动物），230

D

Dalla Gasperina，Marco，146
data compression（数据压缩）。见 efficiency，compression
data hiding（数据隐藏）
and encapsulation（和封装），121
declaration（声明）
vs. definition（与定义），223
definition（定义）
vs. declaration（与声明），223
delete，156，见 new，operator new
dependent names（依赖名字），65
deprecated
meaning of（的含义），20
deque
example use of（的示例用法），222
derivation（派生）
unsafe（不安全），274
derived classes（派生类）
enforcing base class rules（施加基类规则），148
design（设计）
monolithic（巨大的），286，见 monolithic design
destroy（销毁），36
destructor（析构函数）
implicitly generated（隐式生成的），145
destructors（析构函数）。见 RAII
and nofail guarantee（和无失败保证），143
public and virtual，or protected and nonvirtual（公用且虚，或者保护且非虚），136
digraphs，237
Dinkumware，279
disambiguation（消除二义性），225
Disney
allusion to（提及），174
Dog，230
double
conversion to float（转型为 float），229
vs. float（与 float），227

E

e，194
EDG。见 Edison Design Group，53，59，75，278，283
efficiency（效率）
and const（与 const），184，188
compression（压缩），200
premature optimization（不成熟优化），193
string formatting（串格式化），12
using domain-specific knowledge（使用领域相关的知识），210
Ellis，Margaret，70
Employee，27
encapsulation（封装），119，122，274，292，299
breaking（打破），109
encrypting source code（源代码加密），64
endl，7
ends，26
entropy，228
errors（错误）
error handling policy（错误处理策略），82
prefer compile-time to runtime（编译期错误优于运行时错误），150
EvaluateSalaryAndReturnName，87
exception handling（异常处理）
exception specifications（异常规格），89，91，见 exception specifications
exception safety（异常安全）
and try/catch（与 try/catch），80
basic guarantee（基本保证）。见 basic guarantee
nofail guarantee（无失败保证）。见 nofail guarantee
nothrow guarantee（无抛出保证）。见 nofail guarantee
strong guarantee（强保证）。见 strong guarantee
worthwhile（是否值得），85
exception specifications（异常规格），89
a shadow type system（一个浅陋的类型系统），91
and inheritance（与继承），142
and inline（与内联），95
and typedef（与 typedef），92
and virtual functions（与虚函数），93

of implicitly generated functions（隐式生成的函数的），141
exceptions（异常）
translating（转换），83
explicit，260
explicit specialization（显式特化），43
export，59，61，64
and build speed（和构建速度），75
and ease of use（和易用性），74
and Koenig lookup（和 Koenig 查找），73
and macros（和宏），76
and source code organization（和源代码组织），61
guidelines（准则），76
history（历史），69
interaction with other language features（与其他语言特性之间的交互），73
motivation（动机），67
real advantages（实际优点），76
extensibility（可扩展性），276，见 genericity，unintentional loss of

F

FastMemory，169
float
vs. double（与 double），227
floating point（浮点数）。见 float; double
for_each，7
example use（示例用法），28
formatting（格式化）
strings，10
free，156，见 malloc
friends（友元），105，112
and qualification（和名字限定），51
and specializations（和特化），51
and templates（和模板），49
Func，91
function template（函数模板）
specialization（特化），43
function templates（函数模板），42
overloading vs. specialization（重载与特化），42
prefer implementing in terms of class templates（尽量使用类模板来实现函数模板），47
functions（函数）
member vs. nonmember（成员函数与非成员函数），292
nontemplate are best match（非模板函数是最好的匹配），44

G

garbage collection（垃圾收集），153
gcc，53，237，278，283
generic programming techniques（泛型编程技术），1
genericity（泛型性）
unintentional loss of（无意间损失）。36，253，259，274，见 extensibility
Gnu。见 gcc
Google，237

H

header files（头文件）
spelling of（正确拼写），249
heat death，universal（宇宙热寂），228
Heinlein，Robert，248
Hijack（劫持），106
Hunc，91

I

IllegalImmoralAndFattening，270
implicit int（隐式 int）
and main（以及 main 函数），249
implicitly generated functions suppressing（抑制隐式函数生成），147
inheritance（继承）
avoid overusing（避免滥用），103
inline（内联），190
across languages（跨语言），195
and exception specifications（和异常规格），95
at application installation time（在应用程序安装时），195
at coding time（在编码时），192
at compile time（在编译期），193
at JIT time（在 JIT 时），197
at link time（在连接期），194
at run time（在运行时），196
compilers better at judging benefit of（在判断内联的好处上编译器更能胜任），193
function vs. function call（函数与函数调用），193
only a hint（inline 关键字只是一个提示），193
vs. macros（与宏），191
_INT，278
Intel，53，278，283
invalid pointer（无效指针）。见 deleted object
istream_iterator
example use of（示例用法），222
Italian（斜体）
gratuitous use of（无理由的使用），232
iterator range（迭代器区间）
destroy and（销毁和），36
iterators（迭代器）
comparing(with !=，not <)（使用!=而不是<来进行迭代器比较），6
const_iterator，6
operator*，37
pointers are（指针是迭代器），36

J

Java，65，69，73，195，197
Junc，91

K

KerberosAuthenFail，83
keywords（关键字）
auto。见 auto
list of（关键字列表），217
rationale for making reserved words（为什么要使用保留字），214
register。见 register
Koenig lookup（Koenig 查找），73
Koenig，Andrew，179

L

Latin
gratuitous use of（无理由的使用），12，77，175，179，260
lazy allocation（迟分配）。见 allocation，lazy
length-unchecked functions（不检查长度的函数）
never use（永远不要使用这类函数），18
lexical_cast，11，22
lifetime（生命期）
of objects（对象），101
lint，13
Linux，176
_LIST，278

M

Machiavelli，Niccolo

allusion to（提及），108
macros（宏）
abuses of（的滥用），106
and export（和 export），76
controlling（和控制），76
evils of（邪恶的），232
naming（和名字），278
vs. inline（与内联），191
vs. namespaces（与名字空间），234
main（主函数）
returns int（返回 int），249
make_callback，265
malloc，156，见 allocation
Manley，Kevin，284
max munch（"贪婪匹配"），239，243
max_align，272
mem_fun，27
mem_fun_ref，28
member function pointers（成员函数指针）。见 pointers，to member functions
member functions（成员函数）
vs. nonmember functions（与非成员函数），292，310
memory（内存）
committing backing store（提交），176
footprint（占用）。见 allocation
physical（物理），176
virtual（虚拟），176，179
memory exhaustion（内存耗尽）。见 allocation
memory management（内存管理），151，见 allocation strategies（策略），153
memset，178
Metrowerks，53，278，283
Meyers，Scott，169，253，292
Microsoft，53，96，195，278，283
monolithic design（巨类设计），259，286
multimap
example use of（示例），256
Munch，Max，239
Murphy，Capt. Edward A.
allusion to（提及），108
MYUNION，273

N

name lookup（名字查找），114，169
dependent names（依赖名字），65
namespace pollution（名字空间污染），299
namespace（名字空间），250
friend template in a different namespace（位于其他名字空间中的友元模板），50
unnamed namespace（匿名），73
vs. macros（与宏），234
narrowing conversions（窄化转换），229
.NET，65，69，130，195
new，156，见 allocation
conformance issues（与标准的符合问题），177
in-place（定位 new），165
nothrow（nothrow new），165，174，175
operator new。见 operator，new
placement（定位 new）。见 new，in-place
No（不），111
satisfaction（满意），111
nonmember functions（非成员函数）
vs. member functions（与成员函数），292，310
Nonvirtual Interface pattern（非虚接口模式），132
nothrow guarantee（不抛异常保证）。见 nofail guarantee
numeric_limits
example use of（示例），206
NVI。见 Nonvirtual Interface pattern

O

object layout（对象布局）
abusing（误用），107
object lifetime（对象生命期），41，101
ODR（惟一定义规则），61
and export（与 export），74
One Definition Rule（惟一定义规则）。见 ODR
operator（操作符）
()，261
--，prefer prefix form（尽量使用前置形式），6
++，prefer prefix form（尽量使用前置形式），6
=，143
new，156，165，见 allocation
new，and name hiding（new 与名字隐藏），168
new，class-specific（特定类的 new），167
new，replacing（替换 new），166，170
operator--。见 --
operator new。见 operator，new
operator[]
vector。见 vector，operator[]
operator++。见 ++
operators
&，37，241
&&，241
*，37
?:，243
|，242
||，242
+，240
++，240
+=，304
<，242
<<，242
optimization（优化）。见 efficiency
order of construction（构造顺序），100
Orwell，George（乔治·奥威尔）
Animal Farm（动物庄园），10
out_of_range，3
overload resolution（重载决议），114
and export（与 export），73
overloading（重载）
function templates and（函数模板和），44
of primary function templates（主模板），44
prefer to specialization（尽量使用特化），47
specializations not considered（不考虑的特化），44
vs. specialization（与特化）。见 function templates，overloading vs. specialization

P

pair，126
pair2nd，256
parameters（形参）
avoid passing by const value（避免按 const 值传递），189
partial specialization（偏特化），43
pathological code（病态代码），178
Peekaboo parameters（"皮卡布"形参）

见 standard library，Signatures of member functions
persecuted pariahs（被“贬”的设施）
strstream，20
perversity（反常），158
pi，194
Pimpl idiom（Pimpl 惯用法），63
plain old data（旧式数据结构）。见 POD
POD，178
pointer（指针）
invalid（无效指针）。见 deleted object to function（函数指针），92
pointers（指针）
iterators aren't always（迭代器并不总是），36
to member functions（成员函数），29
policy-based design（基于策略的设计），259
polymorphism（多态）
and encapsulation（和封装），121
combining static and dynamic（动静结合），263
compile-time vs. run-time（编译期与运行时），31
preincrement（前置递增）
prefer to postincrement（比后置递增优先），253
premature optimization（不成熟优化）
见 efficiency，premature optimization
PrettyFormat，10
primary template（主模板），43
protected data（保护数据）
why evil（为什么不好），123
public data（公用数据）
why evil（为什么不好），123
public member functions（公用成员函数）
prefer to make nonvirtual（尽量让公用成员函数是非虚的），129
Puppy。见 Frankie

R

RAII（资源获取即初始化），82，86
recursion（递归）。见 recursion
reference（引用）
circular（循环）。见 circular reference
register，219
reserve
vector。见 vector，reserve
reserved words（保留字）。见 keywords
resize
vector。见 vector，resize
resource management（资源管理），151
resuse（复用）
of standard library components（标准库组件的），252

S

security（安全），310
security issues（安全问题）。见 buffer overrun
semicolon（分号）
at end of for statement（语句结尾的），236
separate compilation（分离式编译）
of functions（函数的），63
of templates（模板的）。见 export
Session（会话），83
shared_ptr
example use of（示例），264
prefer instead of auto_ptr（尽量用 shared_ptr 替换 auto_ptr），145
size
vector。见 vector，size
Sleep
and SleepEx，230
SleepEx，230
snprintf，11，16
nonstandard variants（非标准的变种），17
_snprintf，17
sort_idxtbl，246
sort_idxtbl_pair，246
source code（源代码）
encrypting（加密），64
specialization（特化）
and friends（和特化），51
befriending in a different namespace（在不同名字空间中的实体之间建立友元关系），55
explicit，43
of class template（类模板的），43
of function template，43
partial（偏特化），43
prefer overloading instead（尽量使用重载而不是特化），47
vs. overloading（与重载）。见 function templates，overloading vs. specialization
specializations don't overload（特化不参与重载），46
sprintf，10
never use（永远不要使用 sprintf），18
Square，191
Stack，87
standard library（标准库）
implementation latitude（实现的自由度），30
signatures of member functions（成员函数签名问题），29
Star Trek
gratuitous reference to，230
Starbucks
gratuitous advertisement for，129
Stepanov，Alexander，71
string。见 basic_string
string formatting（串格式化），10
_STRING，278
stringstream，11，18
strong guarantee（强保证），85
Stroustrup，Bjarne，xiv，13，70，116，123，205
strstream，11，20
subtle mistakes（微妙的错误）
printf formatting strings（printf 格式化字符串），14
swap，38，40
exception safety（异常安全性），38
specializing or overloading for your own types（为你自己的类型特化或重载），39，40

T

tails（尾巴）
twisting compilers'（扭住编译器的），55
templatability（模板亲和性），14
template，primary（主模板），43
templates（模板）
advantages of befriending with <...> syntax（使用“<...>”这一语法来声明模板特化为友元的好处），58
and friends（和友元），49
and source code organization（和源代码组织），61
cascading recompilation（依次重编译），66
class（类）。见 class templates
dependent names（依赖名字），65

export。见 export
function（函数）。见 function templates
history（历史），70
inclusion model（包含模式），59，60
member templates and access abuses（和滥用访问权限），108
separation model（分离模式），61
separation model(export)（分离模式(export)），59
source code exposure（源代码暴露），62
source dependencies（源代码依赖），62
Turing-complete（图灵完备），70
terminate（中止），94
tests（测试）
unit tests（单元测试），206
throw（抛出），81，83，179
tokenizing（标记化）。见 max munch
transform
example use of（示例），256
trigraphs（三符组）
evils and surprises of（的阴暗面），236
try，80，见 exception safety
Turing-completeness（图灵完备）
and templates（和模板），70
Twice（两次），110
type safety（类型安全性），13
typename
example use of（示例），266

U

unary_function，136，261
unexpected，90，94
unions，269
discriminated（可识别 union），282
uniontype，278
unit tests（单元测试），206

V

Van Winkel，Jan Christiaan，100
Variant，282
vector，2
at，3
capacity，4
iterator not necessarily a pointer（迭代器并不一定是指针），6
operator[]，3
reserve，4
resize，5
size，5
vector<bool> specialization，208
versioning（版本问题）
and robustness（和健壮性），133
vexing parse（恼人的解析）。见 declaration，vs. definition
virtual functions（虚函数）
calling nonvirtually（非虚调用），197
prefer to make nonpublic（尽量做成非公用的），129
visibility（可见性）
vs. accessibility（与可访问性），113，116

W

Webster，Noah，120
Weevil，230
whitespace（空白）。见 auto; register
width specifiers（宽度指示）
minimum vs. maximum（最小与最大），12
WinFX Design Guidelines（WinFX 设计准则），131

X

Xyzzy，32

[General Information]
书名=Exceptional C++ Style中文版
作者=（美）萨特（Sutter,H.）著 刘未鹏译
SS号=11620908
加密地址=
页数=276
下载位置=

书名
版权
前言
目录
正文