

华中科技大学出版社



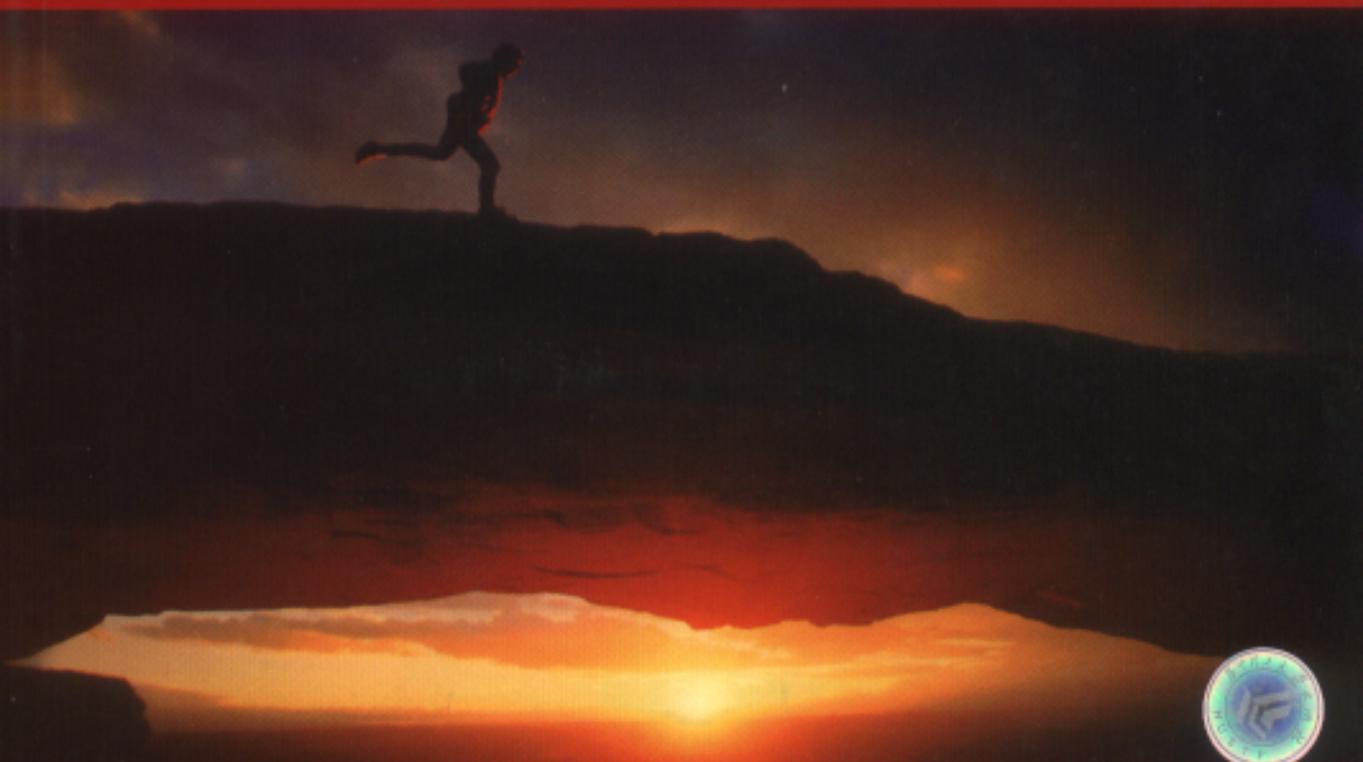
More Exceptional C++ 中文版

40个新的工程难题、编程疑问及解决方法

Herb Sutter 著

於春景 译

肖翔 审校



C++ In-Depth Series • Bjarne Stroustrup

目 录

与大师面对面（译序）	iii
序	v
前言	vii
泛型程序设计与 C++ 标准库	1
条款 1：流	1
条款 2：Predicates，之一：remove() 删除了什么？	6
条款 3：Predicates，之二：状态带来的问题	11
条款 4：可扩充的模板：使用继承还是 traits？	19
条款 5：typename	32
条款 6：容器、指针和“不是容器的容器”	36
条款 7：使用 vector 和 deque	46
条款 8：使用 set 和 map	53
条款 9：等同的代码吗？	59
条款 10：模板特殊化与重载	64
条款 11：Mastermind	69
优化与性能	83
条款 12：内联	83
条款 13：缓式优化，之一：一个普通的旧式 String	86
条款 14：缓式优化，之二：引入缓式优化	90
条款 15：缓式优化，之三：迭代器与引用	94
条款 16：缓式优化，之四：多线程环境	103
异常安全议题及技术	115
条款 17：构造函数失败，之一：对象生命周期	115
条款 18：构造函数失败，之二：吸收异常？	119
条款 19：未捕获的异常	126
条款 20：未管理指针存在的问题，之一：参数求值	132

条款 21: 未管理指针存在的问题, 之二: 使用 auto_ptr?	135
条款 22: 异常安全与类的设计, 之一: 拷贝赋值	141
条款 23: 异常安全与类的设计, 之二: 继承	149
继承与多态	155
条款 24: 为什么要使用多继承?	155
条款 25: 模拟多继承	159
条款 26: 多继承与连体双婴问题	162
条款 27: (非) 纯虚函数	167
条款 28: 受控的多态	172
内存及资源管理	175
条款 29: 使用 auto_ptr	175
条款 30: 智能指针成员, 之一: auto_ptr 存在的问题	182
条款 31: 智能指针成员, 之二: 设计 ValuePtr	187
自由函数与宏	201
条款 32: 递归声明	201
条款 33: 模拟嵌套函数	206
条款 34: 预处理宏	215
条款 35: 宏定义	218
杂项议题	223
条款 36: 初始化	223
条款 37: 前置声明	226
条款 38: typedef	228
条款 39: 名字空间, 之一: using 声明和 using 指令	231
条款 40: 名字空间, 之二: 迁徙到名字空间	234
后记	245
附录 A: (在多线程环境下) 并非优化	247
附录 B: 单线程 String 实现与多线程安全 String 实现的对比测试结果	263
参考文献	271
索引	273

与大师面对面（译序）

小提琴家穆特录制过一张唱片，收录的是贝多芬最伟大的两部小提琴作品^①。聆听那张唱片，你仿佛听见贝多芬在向你倾诉他对音乐艺术的理解和感悟，为你解答小提琴音乐创作的每一个疑问和困惑。为了向音乐爱好者推荐贝多芬的那两部名作，穆特为她的那张名碟加上了一个恰如其分的标题——“与贝多芬面对面”。

合上 *More Exceptional C++* 的瞬间，我的第一反应是为这本名著也加上一个与之类似的副题，然后，郑重地推荐给你。

是的，本书奉献给你的是又一位大师苦心孤诣的结晶。不同的是，这位大师来自你所关注的 C++ 程序设计领域。对每一位 C++ 爱好者或 C++ 专业程序员来说，Herb Sutter 的名字应该不会让人感到陌生。作为 ISO/ANSI C++ 标准委员会的委员，Herb Sutter 不仅是 C++ 程序设计领域公认的专家，还是深受程序员喜爱的技术讲师和作家。Herb Sutter 在互联网上主持的 *Guru of the Week* 专栏广受欢迎，几乎成为每一位 C++ 程序员的网上必读。本书就是 *Guru of the Week* 的最新总结和精华。

本书的主要特点可以反映在我为它所加的副题中。作为身经百战的专业程序员，而且长期从事程序设计的普及教育工作，Herb Sutter 最清楚程序员在提升技术功力的过程中所想所为。他既了解初学者的疑问和困惑，也对专业程序员日常工作中遇到的陷阱和易犯的错误了如指掌。因而，在本书中，Herb Sutter 采用了独具匠心的“提问/解答”的方式来指导你学习 C++ 的语言特性；在本书的每个专题中，Herb Sutter 合理地设想出你的疑问和困惑，又有如神助地猜到了你的（可能是错误的）解答，然后给你以指点并呈现出最佳方案，最后，还提炼出解决类似问题的一般性原则。读了这本书，你仿佛和 Herb Sutter 这位大师面对面地进行了一场对话，亲历了他对你的指导和点拨。

关于本书的另一个特点，我想指出的是，作为 C++ 标准委员会的委员，Herb Sutter 在本书中强调了 C++ 语言的最新标准和最新特性，强调了泛型程序设计和标准库的使用。在本书的所有示例中，Herb Sutter 为我们提供的是清新的 C++ 编程风格和纯正的现代 C++ 代码和范例。

^① 确切地说，这是一套双 CD 专辑，收录的是贝多芬著名的 D 大调小提琴协奏曲和 C 大调“三重奏”协奏曲。

本书适合的读者对象是中高级程序员，但这并不是说本书的内容高不可攀。作者并不是在讲述高深的语言特性和设计技巧，更没有对基础话题避而不谈；相反，有关基础知识的介绍、回顾和深化散见于本书的很多条款之中。只不过，这些条款的内容涉及的只是 C++ 特性中的细节，它没有对 C++ 的每一个特性、或每个特性中的每一个方面作全面的介绍。尽管如此，只要具备基本的 C++ 功底和一定的程序设计经验，你完全可以理解并消化本书的所有内容。由于作者巧妙地组织和精心地选材，本书每一个条款的内容都独立而完整，它让你在深入学习 C++ 语言特性时事半功倍。

因而，我相信，无论是有一定基础的 C++ 爱好者，还是身经百战的专业程序员，本书都会为你带来收获；在你的 C++ 程序设计生涯中，它将是你的案头不可或缺的 C++ 专著之一。

致谢

能够翻译完这本书，我首先要感谢我的妻子姐姐和儿子斗斗，是你们给了我工作的动力，长期以来我都未能陪伴在你们身边，你们不但给了我宽容，还依然给我不变的爱和支持。感谢 yeka，是你促成了我和华中科技大学出版社的愉快合作，并给了 I 最直接的帮助。感谢肖翔对译稿进行的认真审校，他在审阅过程中的真知灼见令我受益匪浅。感谢 moonsea，即使是在我工作的时候，你也会不时地扰乱我的心绪，迫使我无法继续工作——正因为这样，我才得以不时地逃离枯燥的键盘和屏幕，偷得一刻闲暇。感谢作者 Herb Sutter，在翻译本书的过程中，你对我的每一次请教都给予了耐心的解答——和大师的直接对话让我如沐春风。最后，感谢互联网上所有给予我帮助和鼓励的朋友们，lostmouse（我在专业论坛上的网名）希望，这本译作能成为我对你们最好的回馈；我还希望，你们能一如既往地帮助我，指出我在这本译作中留下的每一个疏漏和错误。感谢你们！

於春景 (lostmouse)

2002 年 5 月于深圳蛇口

序

怎样才能成为专家？在我涉足过的所有领域，答案都一样：

1. 掌握基础知识。
2. 将相同的内容再学习一遍，但这一次，请将你的注意力集中在细节上——这些细节的重要性，你头一次可能并没有认识到。

如果挑选了合适的细节来学习，并且彻底掌握了它们，进而达到不再为之困惑的程度，你就离成为专家为期不远了。然而，除非已经成为了专家，你又怎么知道该挑选哪些细节来学习呢？如果有人已经为你挑选了合适的细节，你就会学得更快，并且乐在其中。

举个例子，我曾经参加过一次摄影学习班，授课的是一位很不错的摄影师，名叫 Fred Picker。他告诉我们，摄影中仅有的两个困难的环节是：照相机该摆放在哪儿、何时该按快门。然后，他花了学习班的大部分时间教我们有关曝光、加工和冲印的技术细节——只有完全了解了这些细节，我们才能够很好地掌握摄影；而只有很好地掌握了摄影，我们去关注那两个“困难的”环节才有意义。

学习 C++ 编程的细节，有一个特别引人入胜的方式，即：尽力去回答有关 C++ 编程的问题。例如：

- `f(a++);` 和 `f(a); ++a;` 的效果一样吗？
- 可以用迭代器去改变 `set` 的内容吗？
- 假设你正在使用一个名为 `v` 的 `vector`，它占用的内存数量已经增长到让你担心的程度。于是你会想到去清除这个 `vector`，将内存返还给系统。调用 `v.clear()` 可以完成这一使命吗？

你可能已经猜到，这些表面上看起来显而易见的问题，其答案一定是 No——否则我也不会把它们提出来——但你知道答案为什么是 No 吗？你确信？

本书回答了这些问题。此外，它还回答了其它许多精心挑选的问题，这些问题针对的都是看起来很平常的程序。没有多少书籍具有本书这样的特色——当然，它的前任

*Exceptional C++*除外。有许多自封“高级”的C++书籍，其实它们中的大多数要么只是针对专项主题进行了讨论——如果你只是想精通那些特定主题，而不是想深入研究日常编程中遇到的问题，那些书还不错——要么只是用“高级”这个词来引诱读者而已。

一旦透彻理解了这些问题和它们的答案，你在编程时就不必劳神于细节；你就尽可以将注意力集中在真正需要尽力解决的问题上。

Andrew Koenig

2001年6月

前　　言

古希腊哲学家苏格拉底通过向学生提问进行教学——他用那些精心构思的问题来指导学生，启发他们从已知得出结论；让他们认识到自己正在学习的事物如何相互联系，这些事物与他们已有的知识又如何联系。这种教学法如此出名，以致我们今天把它专称为“苏格拉底问答法”。如果让我们也从学生角度来看问题，苏格拉底的教学法将引导我们，促使我们思考，帮助我们联系并应用现有的知识去获取新知。

本书如同它的前任 *Exceptional C++* [Sutter00]一样，借鉴了苏格拉底的教学法。它假设你目前正身处 C++产品软件开发的某个领域，采用“提问/解答”的形式指导你如何有效地使用标准 C++语言及其标准库——尤其是，如何运用现代 C++中有效的软件工程理论来解决问题。这些问题大多直接取自于我和其他人在编写产品级 C++代码时积累的经验，以所谓的“疑问”和“难题”的形式呈现给你。“疑问”的目的在于帮助你从现有的知识和刚刚学到的知识中得出结论，并展现它们之间如何相互关联。“难题”则向你展示如何去分析 C++设计和编程上的论题——某些是常见的论题，某些并不常见；某些是浅显的论题，某些则比较深奥；还有一些论题之所以拿来讨论，是因为——唔，仅仅因为——它们很有趣。

本书覆盖 C++的方方面面。但我没有说它触及 C++的每一个细节——那将需要更多的篇幅——我是在说，本书提取了 C++语言和标准库构件中的广泛素材，向你展示看似毫不相关的个体如何被综合利用起来，构成解决常见问题的新颖方案。它还展示了这些素材中看似毫不相关的那些部分自身是如何相互关联的——即使有时你不希望它们有如此的关联；以及，应当如何处理这些关联。你将在此找到关于模板与名字空间、异常与继承、健壮类的设计与设计模式、泛型程序设计与宏的使用技巧等内容——这些内容绝不是杂乱的堆砌，而是作为连贯的条款，向你展示现代 C++中这些组成部分之间的相互联系。

何为“More”？

*More Exceptional C++*起步于*Exceptional C++*驻足之处。本书继承了前任的传统：它提供了新的内容，这些内容被组织为短小的条款，形成有主题的章节。前一本书的读者会在此发现一些熟悉的章节和主题，如异常安全、泛型编程、内存管理技术等，但如今它们包含了新的内容。这两本书在结构和主题而非内容上有重叠之处。

*More Exceptional C++*还有何不同之处？本书特别强调了泛型编程技术以及如何有效地使用C++标准库，并涉及了如traits和predicates这样的重要技术。有几个条款还深入分析了使用标准容器和算法时应该牢记的要点——这其中的许多要点，我在别的地方还没见到它们被提及过。此外，一个新的章节和两个附录集中讨论了单线程和多线程环境下的优化议题——对于编写产品级代码的软件开发商来说，这些议题在目前比以往任何时候都更具实用价值。

本书的大多数条款最初出现于互联网和杂志专栏，尤其是Guru of the Week的GotW专题31到62，以及我曾为C/C++ Users Journal、Dr. Dobb's Journal、以前的C++ Report，以及其他出版物撰写的印刷版专栏和文章中。自最初版本出现以来，本书的内容历经大量的修订、增补、校正和更新，因而本书（连同它在www.gotw.ca上不可缺少的勘误表）可以被认为是那些原始材料的最新正式版本。

你应该知道的

我认为你已经了解了C++的基础知识。如果不是这样，建议你从一本好的C++入门和概念性的书籍开始，最好选择一本经典的大部头著作，如Bjarne Stroustrup的*The C++ Programming Language* [Stroustrup00]，或者是Stan Lippman与Josée Lajoie合著的*C++ Primer* 第三版 [Lippman98]。然后，一定要挑选一本指导编程风格的书，例如Scott Meyers经典的*Effective C++*套书[Meyers96] [Meyers97]。我发现这套书有基于浏览器阅读的CD版本[Meyers99]，十分方便好用。

如何阅读本书

本书的每个条款以一个难题或疑问的形式呈现，它带有一条介绍性的标题，类似下面这样：

条款#：条款的题目	难度：X
条款 1：类和对象	困难

条款的题目和难度等级提示你将要面对的是何种难题。注意，难度等级是我的主观评断，我只是猜想大多数人碰到每个问题时会觉得有多难，所以你很可能会发现一个难度为“7”的问题对你来说比某个难度为“5”的问题还容易。自从写作 *Exceptional C++* 以来，我不时收到一些电子邮件，说“某某条款比你说的要容易（难）！”面对同一条款，不同的人认为“更容易！”或“更难！”是很自然的。难度等级随人而定；对你来说，任何条款的实际难度真正取决于你的知识和经验，它对别人来说会相对更容易或更难。但大多数情况下你会发现，难度分级是一种不错的经验方法，它指引给你的和你所认为的大致相当。

你可能打算从头至尾阅读整本书；这很好，但不一定非如此不可。你可能会集中阅读一个章节中的所有条款，因为你对那个章节的议题特别感兴趣；这也不错。书中有一些条款被我称为“短系列”，因为它们涉及的是相关的问题，你会看到这些条款用“之一”、“之二”等等来标识。除了这些“短系列”外，其余的条款都是很独立的。在本书的条款中还包含很多交叉索引，有些索引还参考到 *Exceptional C++*，你可以遵循这些索引随意跳转阅读。我唯一要告诉你的是，制作短系列是为了让它们成为一组以便于连续阅读；除此之外，如何阅读，选择权在你。

名字空间、`typename`、URL 引用以及其它约定

我在本书中提出了不少建议，但我不会指引你去做一些连我自己都没做过的事。这包括整本书里我在我自己的示例代码中所做的那些事。我也会遵循程序设计的现有习惯和现代风格，即使有时候这样做不会对事情带来本质上的差异。

关于这一点，说说名字空间：在本书的代码示例中，如果你在一个例子中看到了一个文件范围内的 `using` 指令，又在几页或几个条款后另外的例子中看到了一个函数范围内的 `using` 指令，这其实没有什么更深层的含义，只是说明在那些特定情况下，这样做是合理的，而且从美学的角度来看，也让我感到美观；至于名字空间的基本知识，请参阅条款 40。在书写代码时，如果想强调我所指的是标准中的东西，我会用 `std::` 来修饰标准库名称。这一点确立后，我往往会转而使用不带修饰的名称。

再说说模板参数的声明。我时常碰到一些人，他们认为写 `class` 而不写 `typename` 是过时的做法，即使这二者没有功能上的不同、而且 C++ 标准本身也在到处使用 `class`。纯

粹出于代码书写风格方面的考虑，并且为了强调本书所讨论的是当今现代的 C++，在声明模板参数时，我也已经转到使用 `typename` 而不使用 `class`。唯一的例外是条款 33 中的一处，在那儿我直接引用了标准中的代码；标准用的是 `class`，我就随它去了。

除非我明确地称某段代码是“完整程序”，否则一般不是。请记住，这些示例通常只是代码片段或者只是程序的一部分，它们不会就这样孤立地通过编译。为了从我所提供的程序片段构成完整程序，你一般还得做一些显而易见的添加工作。

最后，说说 URL：互联网上，事事在变。特别是，你无法掌控的那些事物在变。这样一来，在印刷书籍上随意发布 URL 就成为了一种真正的痛苦：不用等到一本书在你的书桌上躺上五年，在它还没送到印刷厂之前，那些 URL 可能就已经过时了。本书中，当我引用其他人的文章或网站时，我是通过我自己的网站 www.gotw.ca 上的 URL 来实现的；这个网站我可以自己控制，它包含的只不过是直接指向实际网页的重定向链接。如果你发现印刷在本书中的某个链接不再可用，请发电子邮件告诉我：我会更新这个链接，让它指向新的网页位置（如果我能够重新找到这个网页），或者标示这个网页已经不再存在（如果我无法找到这个网页）。无论哪种方式，本书的 URL 将保持最新——尽管在当今互联网世界里，印刷传媒是这样举步维艰。唉！

致谢

深切感谢丛书编辑 Bjarne Stroustrup，还有 Debbie Lafferty、Tyrrell Albaugh、Chanda Leary-Coutu、Charles Leddy、Curt Johnson，以及 Addison-Wesley 出版社的其它成员，感谢他们在这个项目中的鼎力协助和坚持不懈。很难想象还能找到比他们更棒的人一起共事，他们的热情和协作使这本书完全达到了我预想的目标。

另外值得感谢和称赞的是审阅过本书的许多专家。对本书的许多内容，他们毫无保留地提出了深刻的见解和犀利的批评，而这些见解和批评是那样一针见血。正是因为他们努力，你手中的这本书比初稿更完整、更易于理解、更实用。特别感谢（大致以我收到审阅意见的顺序）Scott Meyers、Jan Christiaan van Winkel、Steve Dewhurst、Dennis Manci、Jim Hyslop、Steve Clamage、Kevlin Henney、Andrew Koenig、Patrick McKillen，以及一些不知名的审阅者。书中遗留的所有错误、疏忽和歧义都是因为我，而非他们。

最后，将所有的感谢献给我的家人和朋友——无论是在这本书的写作和出版期间，还是其它任何时候，他们都一直陪伴在我身边。

Herb Sutter
2001 年 6 月于多伦多

泛型程序设计与 C++ 标准库

C++ 威力强大的特性之一是对泛型程序设计 (generic programming) 的支持。这种威力直接反映在 C++ 标准库的灵活性上，特别是它的容器、迭代器和算法部分，这一部分一直以来被称作标准模板库 (STL)。

本书的开篇章节集中讨论如何最有效地使用 C++ 标准库，尤其是 STL。什么时候使用 `std::vector` 和 `std::deque` 会最有效？如何使用？在使用 `std::map` 和 `std::set` 的时候可能会碰到哪些陷阱？如何安全地避免这些陷阱？`std::remove()` 为什么不能真正删除任何东西？

本章还特别介绍了一些有用的技巧和易犯的错误，在撰写自己的泛型程序代码的时候，包括撰写那些“用以和 STL 一起工作”或“用以扩充 STL”的代码的时候，你会经常碰到它们。什么样的 predicates 才能安全地和 STL 一起使用？什么样的不行？为什么？要想让模板自身的行为能够被改变，而且这种“行为的改变”是基于“与模板协同工作的类型 (type)”的能力，有什么现有的技术可以写出这种功能强大的泛型模板代码吗？如何在不同种类的输入输出流之间自如地切换？模板特殊化和重载是怎么一回事？古怪的 `typename` 关键字究竟有何过人之处？

随着对泛型程序设计和 C++ 标准库有关话题的深入研究，我们还会碰到更多的问题。

条款 1：流

难度：2

在动态地使用不同的输入输出流——包括标准控制台流 (`console stream`) 和文件流，最佳使用方式是什么？

(1) `std::cin` 和 `std::cout` 的类型是什么？

(2) 写一个 ECHO 程序，让它简单地响应输入，并能通过以下两种方式等效地调用：

```
ECHO <infile >outfile
```

```
ECHO infile outfile
```

在大多数流行的命令行环境下，第一个命令假定程序从 `cin` 获得输入，并将输出发送到 `cout`。第二个命令告诉程序从一个名为 `infile` 的文件中获得输入，并在名为 `outfile` 的文件中产生输出。这个程序应该能够支持以上所有的输入/输出选项。

解答

1. `std::cin` 和 `std::cout` 的类型是什么？

简短地回答，`cin` 实际上是：

```
std::basic_istream<char, std::char_traits<char>>
```

`cout` 实际上是：

```
std::basic_ostream<char, std::char_traits<char>>
```

下面是较详细的回答，它通过一些标准的 `typedef` 和模板向你展示答案的来龙去脉。首先，`cin` 和 `cout` 具有的类型分别是 `std::istream` 和 `std::ostream`。接着，这些类型是 `std::basic_istream<char>` 和 `std::basic_ostream<char>` 的 `typedef`。最后，考虑到模板参数的默认值，我们得到上面的答案。

注意：如果你使用的 `iostream` 子系统是 C++ 标准制定之前的实现版本，你可能还会看到一些中间类（intermediate class），例如 `istream_with_assign`。但这些类在标准中是不存在的。

2. 写一个 ECHO 程序，让它简单地响应输入，并能通过以下两种方式等效地调用：

```
ECHO <infile >outfile
```

```
ECHO infile outfile
```

最精简的方案

对于追求精简代码的人来说，最精简的方案莫过于下面这个程序，它仅包含一条语句：

```

// 例 1-1：惊讶吗？只使用了一条语句
//
#include <fstream>
#include <iostream>

int main( int argc, char* argv[] )
{
    using namespace std;

    (argc > 2
     ? ofstream(argv[2], ios::out | ios::binary)
       : cout)
      <<
    (argc > 1
     ? ifstream(argv[1], ios::in | ios::binary)
       : cin)
      .rdbuf();
}

```

这个方案之所以可行，得益于两个相辅相成的条件：第一，`basic_ios` 提供了一个方便的 `rdbuf()` 成员函数，它返回某个流对象所使用的 `streambuf`，在本例中，这个流对象也就是 `cin` 或临时 `ifstream` 对象，二者都派生于 `basic_ios`。第二，`basic_ostream` 提供了一个 `operator<<()`，它正好接受这样的 `basic_streambuf` 对象，将其作为输入，然后完全读取输入。正如法国人会说的那样，“C'est ça”（“就是这样！”）。

逐步趋向更灵活的方案

例 1-1 中的方案有两个主要缺点：首先，精简会带来晦涩，而且过度的精简不适合应用到产品代码中。

设计准则

尽量提高可读性。避免撰写精简代码（即，简洁但难以理解和维护）。避免晦涩。

第二，虽然例 1-1 回答了前面的问题，但只是在对输入进行逐字拷贝的情况下，这种方法才可行。这种功能在目前可能已经够用，但如果将来你需要对输入进行其它处理，例如将字符转换成大写，或是计算字符总数，或删除第三个字符，那该怎么办？这种需要在将来是很合理的；所以，最好我们现在就立即动手，将这些处理工作封装在一个单独的函数中，使这个函数可以多态地（*polymorphically*）使用正确类型的输入或输出对象：

```

#include <fstream>
#include <iostream>

int main( int argc, char* argv[] )
{
    using namespace std;

    fstream in, out;
    if( argc > 1 ) in.open( argv[1], ios::in | ios::binary );
    if( argc > 2 ) out.open( argv[2], ios::out | ios::binary );

    Process( in.is_open() ? in : cin,
             out.is_open() ? out : cout );
}

```

但如何实现 Process()? 在 C++ 中，主要有四种方法获得多态行为：虚函数、模板、重载和转换。其中，前两种方法可以直接用在这里，用以表达我们所需要的多态。

方法 A：模板（编译时多态）

第一种方法使用的是编译时多态，这需要借助于模板；它只需要被传递的对象有一个合适的接口（例如一个名为 rdbuf() 的成员函数）：

```

// 例 1-2(a): 模板化的 Process()
//
template<typename In, typename Out>
void Process( In& in, Out& out ) {
    // ... 执行某种更复杂的操作,
    //      或只是简单的 "out << in.rdbuf();" ...
}

```

方法 B：虚函数（运行时多态）

第二种方法使用的是运行时多态，它需要一个条件，即，存在一个具有合适接口的公共基类：

```

// 例 1-2(b): 第一次尝试，一定程度上可行
//
void Process( basic_istream<char>& in,
              basic_ostream<char>& out )
{
    // ... 执行某种更复杂的操作,
    //      或只是简单的 "out << in.rdbuf();" ...
}

```

注意，在例 1-2(b)中，Process()的参数类型不是 `basic_ios<char>&`，因为那将不允许使用 `operator<<()`。

毫无疑问，例 1-2(b)中的方法具有依赖性，它要求输入和输出流必须分别从 `basic_istream<char>` 和 `basic_ostream<char>` 派生。这一点对我们的例子来说还不错，但要知道，并非所有的流都基于简单的 `char` 或者 `char_traits<char>`。例如，宽字符流基于 `wchar_t`，*Exceptional C++* [Sutter00] 的条款 2 和 3 也演示了一些具有不同行为特征的用户自定义 traits（在那些例子中，`ci_char_traits` 提供了大小写不分的行为特征），并展示了其潜在的用途。

因而，即使是采用方法 B，我们也应该使用模板，让编译器去推导出合适的参数：

```
// 例 1-2(c): 更好的方案
//
template<typename C, typename T>
void Process( basic_istream<C,T>& in,
              basic_ostream<C,T>& out )
{
    // ... 执行某种更复杂的操作,
    //      或只是简单的 "out << in.rdbuf();" ...
}
```

有效的工程设计原则

就其本身而言，以上所有答案都是“正确”的；但在目前场合下，我个人倾向于选择方法 A。其原因归结于两条很有价值的设计准则。第一条是：

设计准则

尽量提高可扩充性。

避免写出的代码只能解决当前问题。几乎任何时候，若能写出可扩充的方案，那将是更佳选择——当然，只要我们不太过分。

均衡的判断力是有经验的程序员所具有的一个特征。尤其是，在“编写专用代码，只解决当前问题”（短视，难以扩充）和“编写一个宏大的通用框架去解决本来应该很简单的问题”（追求过度设计）之间，有经验的程序员懂得如何去获取最佳的平衡。

较之例 1-1 中的方案，方法 A 具有大致相同的整体复杂度，但除此之外，后者还更容易理解、更具可扩充性。较之方法 B，方法 A 既简单又更具灵活性；它更能适应新的要求，因为它没有了束缚，不只是能和 `iostream` 体系打交道。

所以，如果存在两个选择，它们在设计和实现中需要的工作量相同，而且具有大致相当的清晰度和可维护性，那么，请尽量考虑可扩充性。这条建议并不是在教唆你，让你去对一个本来很简单的问题大动干戈——这方面大家以前已经做得够多了。相反，这条建议是一条鼓励：如果稍微思考一下就可以发现，自己正在解决的问题实际上是某个更通用的问题的特例，你就应该多做一些工作，而不要仅仅满足于解决当前问题。这条建议十分正确，因为在设计中提高了可扩充性，往往意味着同时提高了封装性。

设计准则

尽量提高封装性。将关系分离。

只要有可能，一段代码——函数或类——应该只知道并且只负责一件事。

可以证明，方法 A 最出色的地方在于：它展示了关系之间最好的分离。它包括两部分代码，一部分代码知道输入/输出源（source）和目标（sink）中可能的区别，另一部分代码知道如何真正执行处理，这两部分代码被分离开来。这种分离也使得代码的用途更清晰，更易于他人阅读和理解。将关系进行最佳分离是好的工程设计的另一个特征，在本书条款中，我们将不断地看到这一点。

条款 2: Predicates, 之一: remove()删除了什么?

难度: 4

本条款可以测试你使用标准算法的能力。标准库算法 `remove()` 实际完成什么功能？如何着手去写一个泛型函数，用以删除一个容器中的第三个元素？

(1) `std::remove()` 算法完成什么功能？请明确回答。

(2) 写一段代码，用来删除 `std::vector<int>` 中值等于 3 的所有元素。

(3) 为了删除一个容器中的第 n 个元素，你的开发小组中的某位程序员写了下面两段代码供你选择。

```
// 方法 1: 写一个专用的
// remove_nth 算法
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
    /* ... */
}
```

```

// 方法 2: 写一个函数对象, 当它被
// 使用 n 次后返回 true; 将这个函数
// 对象用作 remove_if 的 predicate
//
class FlagNth
{
public:
    FlagNth( size_t n ) : current_(0), n_(n) { }

    template<typename T>
    bool operator()( const T& ) { return ++current == n_; }

private:
    size_t current_;
    const size_t n_;
};

// 调用示例
... remove_if( v.begin(), v.end(), FlagNth(3) ) ...

```

- a) 实现方法 1 中缺少的那部分代码。
 b) 哪种方法更好? 为什么? 就两种方案中可能出现的任何问题进行讨论.

练习

remove()删除了什么?

1. std::remove()算法完成什么功能? 请明确回答.

从物理意义上来说, 标准算法 `remove()`没有将对象从容器中删除: `remove()`执行完后, 容器的大小不变. 更确切地说, `remove()`只是简单地用“未删除”对象来填补被删除对象留下的缺口, 每一个被删除对象在尾部还是会留下一个相应的“死亡”对象. 最后, `remove()`返回一个迭代器, 指向第一个“死亡”对象; 或者, 如果没有对象被删除, `remove()`将返回 `end()`迭代器.

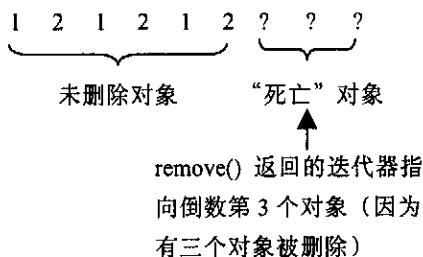
例如, 请看这样一个 `vector<int>` `v`, 它包含下面 9 个元素:

1 2 3 1 2 3 1 2 3

假设用下面的代码来删除容器中所有值为 3 的元素：

```
// 例 2-1
// remove( v.begin(), v.end(), 3 ); // 微妙的错误
```

结果将会如何？答案大致像下面这样：



三个对象必须得被删除，剩余的对象会被拷贝，以填充它们留下的缺口。容器尾部的对象可能还是保持原始值（1 2 3），也可能不是——所以不要相信这一假设。再次注意，容器的大小没变。

你一定会奇怪：remove()为什么要以这样的方式工作。其根本原因在于：remove()不是作用在容器身上，而是作用在迭代器区间（range）上，“从任何一种容器中删除迭代器所指元素”这样的迭代器操作是不存在的。要想这样做，我们必须真正、直接地得到容器。关于 remove()更进一步的知识，请参见 Andrew Koenig 在[Koenig99]中就这一主题所作的全面论述。

2. 写一段代码，用来删除 std::vector<int> 中值等于 3 的所有元素。

下面这行简短的代码将不辱使命，其中 v 是一个 vector<int>：

```
// 例 2-2: 从 vector<int> v 中删除所有的 3
v.erase( remove( v.begin(), v.end(), 3 ), v.end() );
```

对 remove(v.begin(), v.end(), 3) 的调用将完成实际工作，并返回一个迭代器，指向第 1 个“死亡”元素。然后，在第 1 个“死亡”元素所在处直至 v.end() 的区间上调用 erase()，从而删除所有死亡元素，使得 vector 只包含未被删除的对象。

3. 为了删除一个容器中的第 n 个元素，你的开发小组中的某位程序员写了下面两段代码供你选择。

```

// 例 2-3(a)
//
// 方法 1: 写一个专用的
// remove_nth 算法
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
    /* ... */
}

// 例 2-3(b)
//
// 方法 2: 写一个函数对象, 当它被
// 使用 n 次后返回 true; 将这个函数
// 对象用作 remove_if 的 predicate
//
class FlagNth
{
public:
    FlagNth( size_t n ) : current_(0), n_(n) { }

    template<typename T>
    bool operator()( const T& ) { return ++current == n_; }

private:
    size_t current_;
    const size_t n_;
};

// 调用示例
... remove_if( v.begin(), v.end(), FlagNth(3) ) ...

```

a) 实现方法 1 中缺少的那部分代码.

人们给出的实现往往有下面代码中那样的臭虫. 你也未能幸免吗?

```

// 例 2-3(c): 你能发现问题所在吗?
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
    for( ; n > 0; ++first, --n )
    {
        if( first != last )
        {
            FwdIter dest = first;
            return copy( ++first, last, dest );
        }
    }
    return last;
}

```

例 2-3(c) 存在一个问题，且一个问题涉及到两个方面：

(1) 正确的前提：我们没有要求 $n \leq \text{distance}(\text{first}, \text{last})$ ，所以第一个循环语句可能会使 `first` 越过 `last`，造成 `[first, last)` 不再是一个有效的迭代器区间。如果是这样，在函数的剩余部分，麻烦就会找上门。

(2) 效率：假设我们想确保（并检查）“`n` 对给定的区间有效”这一前提，将其作为解决第一个问题的方法。那么，我们还是应该完全放弃那个迭代器递增循环，而只需简单地写一句 `advance(first, n)`。标准 `advance()` 算法是为迭代器而设计的，它可以分辨迭代器类别（iterator categories）；针对随机访问迭代器（random-access iterator），它能自动地进行优化。具体来说，针对随机访问迭代器，它只需要消耗常数级时间，而对于其它迭代器，它需要消耗线性级时间。

这里是一个合理的实现：

```
// 例 2-3(d): 解决问题
//
// 前提：
// —— n 不能超过区间的大小
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
    // 检查前提：只是在调试模式下带来开销
    assert( distance( first, last ) >= n );

    // 实际工作
    advance( first, n );
    if( first != last )
    {
        FwdIter dest = first;
        return copy( ++first, last, dest );
    }
    return last;
}
```

b) 哪种方法更好？为什么？就两种方案中可能出现的任何问题进行讨论。

方法 1 有两个主要优点：

1. 它是正确的。
2. 它利用了 iterator traits 特性，特别是迭代器类别，因而在随机访问迭代器身上表现更佳。

方法 2 存在相应的缺点，我们将在本短系列的第二部分详细分析。

条款 3: Predicates 之二: 状态带来的问题

难度: 7

紧接着条款 2 的介绍，我们现在来讨论“状态性” predicate。什么是状态性 predicate？它们何时有用？它们和标准容器及算法的兼容性如何？

- (1) 什么是 predicate？在 STL 中，它们是如何被使用的？给出一个例子。
- (2) “状态性” predicate 何时有用？给出例子。
- (3) 为了让状态性 predicate 正常工作，对算法有哪些必要的要求？

解答

一元和二元 predicate

1. 什么是 predicate？在 STL 中，它们是如何被使用的？给出一个例子。

Predicate（谓词）是一个函数指针或一个函数对象（function object）（一个提供了函数调用运算符 operator() 的对象）；针对一个“关于对象的提问”，它作出“是”或“否”的回答。许多算法使用 predicate，用以对它们所操作的每个元素进行某种判断；所以，当像下面这样使用时，这个名为 pred 的 predicate 会正常运作：

```
// 例 3-1(a): 使用一元 predicate
//
if( pred( *first ) )
{
    /* ... */
}
```

正如你在这个例子中所看到的，pred 应当返回一个值，而且这个值可以用 true 来作检测。注意，predicate 只能通过“被解引用（dereferenced）的迭代器”来使用 const 函数。

有些 predicate 是二元（binary）的——它们取两个对象（通常是被解引用的迭代器）作为参数。这意味着，当象下面这样使用时，这个名为 bpred 的二元 predicate 会正常运作：

```
// 例 3-1(b): 使用一个二元 predicate
//
if( bpred( *first1, *first2 ) )
{
    /* ... */
}
```

给出一个例子。

请看下面的代码，这是标准算法 `find_if()` 的实现：

```
// 例 3-1(c): find_if()示例
//
template<typename Iter, typename Pred> inline
Iter find_if( Iter first, Iter last, Pred pred )
{
    while( first != last && !pred(*first) )
    {
        ++first;
    }
    return first;
}
```

这个算法依次访问区间 `[first, last]` 中的每个元素，并对每个元素调用 `pred`；`pred` 是个 `predicate`，它可以是函数指针或对象。如果存在一个元素，`predicate` 对它所求得的值为 `true`，`find_if()` 将返回一个迭代器，指向第 1 个这样的元素。否则，`find_if()` 返回 `last`，表示没有找到可以满足这个 `predicate` 要求的元素。

我们可以像下面这样，通过一个函数指针 `predicate` 来使用 `find_if()`：

```
// 例 3-1(d):
// 通过函数指针来使用 find_if()
//
bool GreaterThanFive( int i )
{
    return i > 5;
}

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
    return find_if( v.begin(), v.end(), GreaterThanFive )
           != v.end();
}
```

下面是相同的例子，不过这次没有使用自由函数，而是通过函数对象来使用 `find_if()`：

```
// 例 3-1(e):
// 通过函数对象来使用 find_if()
//
class GreaterThanFive
    : public std::unary_function<int, bool>
{
public:
    bool operator()( int i ) const
    {
        return i > 5;
    }
};
```

```
bool IsAnyElementGreaterThanFive( vector<int>& v )
{
    return find_if( v.begin(), v.end(), GreaterThanFive() )
           != v.end();
}
```

在这个例子中，使用函数对象较之使用自由函数没有得到什么好处，是吗？这就刚好将我们引入到另外几个问题中——在那儿，函数对象展示了更大的灵活性。

2. “状态性” predicate 何时有用？给出例子。

接着例 3-1(d)和例 3-1(e)继续往下讨论。对于下面这样的任务，自由函数无法轻松完成——除非借助静态变量之类的东西：

```
// 例 3-2(a):
// 通过更通用的函数对象来使用 find_if()
//
class GreaterThan
    : public std::unary_function<int, bool>
{
public:
    GreaterThan( int value ) : value_( value ) { }
    bool operator()( int i ) const
    {
        return i > value_;
    }
private:
    const int value_;
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
    return find_if( v.begin(), v.end(), GreaterThan(5) )
           != v.end();
}
```

GreaterThan 这个 predicate 有个数据成员，用来记录一个值；在本例中，这个值要和每一个元素做比较。你可能已经看到，这个版本比例 3-1(d)和例 3-1(e)中的专用代码更有用，可复用性更高，而这一切威力都来自“在对象内部存储局部信息的能力”。

进一步，我们最终得到更加通用的版本：

```
// 例 3-2(b):
// 通过完全通用的函数对象来使用 find_if()
//
```

```

template<typename T>
class GreaterThan
    : public std::unary_function<T, bool>
{
public:
    GreaterThan( T value ) : value_( value ) { }

    bool operator()( const T& t ) const
    {
        return t > value_;
    }

private:
    const T value_;
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
    return find_if( v.begin(), v.end(), GreaterThan<int>(5) )
           != v.end();
}

```

可见：使用可以储值的 predicate，可以获得可用性上的某些好处。

进一步：状态性 predicate

例 3-2(a)和例 3-2(b)中的 predicate 都有一个重要的性质：拷贝和原始对象是等同的。也就是说，如果你得到 GreaterThan<int>对象的一个拷贝，那么，这个拷贝的行为在所有方面和原始对象一样，二者可以互换使用。正如我们将在问题 3 中看到的那样，这一性质很重要。

有些人尝试着写出一些所谓的“状态性（stateful）”predicate，这些 predicate 在功能上更进了一步：它们在使用时会发生变化——也就是说，在使用一个 predicate 时，其使用结果依赖于前一次使用的历史情况。在例 3-2(a)和例 3-2(b)中，对象的确携带了一些内部值，但这些值在构造时期就已经固定了；它们不是状态值，在对象的生命期，它们不会发生变化。当谈论状态性 predicate 时，我们主要指的是“具有可变状态”的 predicate——它们对发生在自己身上的事件非常敏感，就像一部小型状态机。^①

在某些书籍中，我们可以看到这种状态性 predicate 的例子。特别是，人们已经尝试着写出这样一种 predicate：当 predicate 被作用到哪些元素上，它们就可以记住那些元素的各种信息。例如，一些 predicate 可以记住它们所作用的对象的值，用以执行计算（例

^① 正如 John D. Hickin 所精彩描述的那样：“输入[first, last)有点像供给图灵机的磁带，状态性 predicate 则像一个程序”。

如，有这样一个 predicate，如果自它被使用以来，它所作用的元素的平均值超过 50，或累计值小于 100，或其它什么条件，这个 predicate 就返回 true）。实际上，我们已经在条款 2 的问题 3 中看到过一个具体的例子，这个例子使用了这种状态性 predicate：

```
// 例 3-2(c)
// (源自条款 2，例 2-3(b))
//
// 方法 2：写一个函数对象，当它被
// 使用 n 次后返回 true；将这个函数
// 对象用作 remove_if 的 predicate
//
class FlagNth
{
public:
    FlagNth( size_t n ) : current_(0), n_(n) { }

    template<typename T>
    bool operator()( const T& ) { return ++current == n_; }

private:
    size_t current_;
    const size_t n_;
};
```

对上面这样的状态性 predicate 来说，它们如何被作用到操作区间的元素上，其方式很重要。本例中的这个状态性 predicate 就特别依赖于两点：它被区间里的元素使用的次数，以及使用的次序（例如，如果它和 remove_if() 之类的算法一起使用的话）。

状态性 predicate 和非状态性 predicate 的主要区别在于：对于状态性 predicate 来说，拷贝和它们本身不是等同的。很明显，一个算法不能复制出一个 FlagNth 对象，然后将一个对象作用在一些元素身上，将另外一个对象作用在另外一些元素身上。这绝对不会带来你所预计的结果，因为这两个 predicate 对象会单独更新它们的计数，它们都不能够标示出正确的第 n 个元素。它们只能分别标示出“它们本身”在被使用时的第 n 个元素。

问题在于，在例 3-2(c) 中，方法 2 可能会以这样的方式使用 FlagNth 对象：

```
// 调用示例
... remove_if( v.begin(), v.end(), FlagNth(3) ) ...
```

“看起来很合理啊，而且我使用过这种技术。”一些人会说。“我正好读过一本 C++ 书籍，书中演示了这一技术，所以一定没错。”一些人会说。可是，事实是：这一技术可能刚好在你的编译器上可行（或者，在那本书的作者所使用的编译器上可行），但你不能保证可以将它移植到所有的编译器、甚至你（或那位作者）正在使用的编译器的下一个版本上。

要知道为什么，让我们在第 3 个问题中对 `remove_if()` 稍作详细分析：

3. 为了让状态性 predicate 正常工作，对算法有哪些必要的要求？

要让状态性 predicate 对算法真正有用，在如何使用 predicate 方面，算法一般得保证两点：

- (a) 算法绝不能对 predicate 做复制（即，自始至终只能使用同一给定对象）。
- (b) 算法必须“以某个已知的顺序”将 predicate 作用到区间里的元素上。（通常是，从第 1 个元素到最后一个元素）

可惜的是，C++ 标准没有要求标准算法提供以上两个保证。虽然状态性 predicate 已经出现在一些书中，但是，在书和标准的决斗中，胜利者最终会是标准。对于标准算法，C++ 标准确实有很多其它规定，例如性能复杂度、predicate 被使用的次数等等，但对于所有的算法，它恰恰没有规定上面(a)中的要求。

例如，请看 `std::remove_if()`：

- (a) 在标准库实现中，`remove_if()` 通常用 `find_if()` 来实现，而且，predicate 是以传值的方式被传递给 `find_if()`。这会使 predicate 的行为不可预测，因为，实际传递给 `remove_if()` 的 predicate 对象不一定会被区间里的每个元素使用一次。相反，一定会被每个元素使用一次的是这个 predicate 对象“或这个 predicate 对象的一个拷贝”。这是因为，一个符合标准的 `remove_if()` 算法可以假设 predicate 的拷贝和 predicate 本身是等同的。
- (b) C++ 标准要求，提供给 `remove_if()` 的 predicate 应该刚好被使用 `last-first` 次，但它没有要求以何种顺序使用。这样，有人就会写出这么一个 `remove_if()` 实现：它符合标准，但没有按次序将 predicate 作用于每个元素；这样的实现固然可恶，但完全是可能的。所以请记住，只要标准没有这样的规定，你就甭想依靠它写出具有可移植性的代码。

那么，你会问，“难道就没有办法让 `FlagNth` 这样的 predicate 可以和标准算法一起可靠地工作吗？”很遗憾，答案是——没有。

没关系，没关系！我可以听到一些人充满怒火地咆哮。为了解决上面问题(a)中的 predicate 拷贝问题，这些人写出了一些运用引用计数技术的 predicate。是的，你可以共享 predicate 状态，这样，在 predicate 被使用时，它就可以被安全地复制，而其语义不会发生变化。下面的代码使用了这一技术（在本例中，此技术用于一个合适的 `CountedPtr` 模板中；附加问题：提供一个合适的 `CountedPtr` 实现）：

```
// 例 3-3(a)：一个（不完整的）方案，  
//           拷贝之间共享状态  
//  
class FlagNthImpl
```

```

{
public:
    FlagNthImpl( size_t nn ) : i(0), n(nn) { }
    size_t      i;
    const size_t n;
};

class FlagNth
{
public:
    FlagNth( size_t n )
        : pimpl_( new FlagNthImpl( n ) )
    {
    }

    template<typename T>
    bool operator()( const T& )
    {
        return ++(pimpl_->i) == pimpl_->n;
    }

private:
    CountedPtr<FlagNthImpl> pimpl_;
};

```

但这没有、也不可能解决前面问题(b)中的次序问题。也就是说，作为程序员，你必须完全依赖于算法使用 predicate 的次序。你无法逃避这一点，即使运用高级的指针技术也无能为力，除非算法本身保证遍历的次序。

附加问题

上面的方案有一个附加问题：提供一个合适的 CountedPtr 实现。CountedPtr 是一个智能指针，它的拷贝和它一样，指向的是相同的实体；最后一个被销毁的拷贝负责清除被分配的对象。下面这个实现是可行的，但要将它用于产品，尚需进一步完善：

```

template<typename T>
class CountedPtr
{
private:
    class Impl
    {
    public:
        Impl( T* pp ) : p(pp), refs( 1 ) { }

        ~Impl() { delete p; }

        T*      p;
        size_t  refs;
    };
    Impl*  impl_;

```

```

public:
    explicit CountedPtr( T* p )
        : impl_( new Impl( p ) ) { }

    ~CountedPtr() { Decrement(); }

    CountedPtr( const CountedPtr& other )
        : impl_( other.impl_ )
    {
        Increment();
    }

    CountedPtr& operator=( const CountedPtr& other )
    {
        if( impl_ != other.impl_ )
        {
            Decrement();
            impl_ = other.impl_;
            Increment();
        }
        return *this;
    }

    T* operator->() const
    {
        return impl_->p;
    }

    T& operator*() const
    {
        return *(impl_->p);
    }

private:
    void Decrement()
    {
        if( --(impl_->refs) == 0 )
        {
            delete impl_;
        }
    }

    void Increment()
    {
        ++(impl_->refs);
    }
};

```

条款 4：可扩充的模板：使用继承还是 traits?

难度：7

本条款讨论了 traits 模板，并示范了一些高级 traits 技术。如果没有 traits，模板能从它的类型了解到哪些信息？又能针对它做些什么？答案俏皮而且很有启发性，其意义不仅仅只针对于开发 C++ 程序库的程序员。

(1) 什么是 traits 类？

(2) 示范如何检测和运用模板参数的成员，请使用下面这个具有启发性的例子：假设你想写一个类模板 C，能够实例化此模板的类型必须具有一个名为 Clone() 的 const 成员函数，此函数不带参数，返回值为指针，指向同种类型的对象。

```
// T 必须提供 T* T::Clone() const
template<typename T>
class C
{
    // ...
};
```

注意到一条很显然的事实：如果在模板 C 中写一段代码，让它去调用不带参数的 T::Clone()，那么，实际上没有这样一个“不带参数就可以被调用”的 T::Clone()，这段代码就不能够通过编译。但这一事实并不足以回答本问题。因为，在调用不带参数的 T::Clone() 时，如果存在一个带缺省参数并且（或者）不返回 T* 的 Clone()，调用也会成功。这里，我们的目标明确限定为：T 提供的函数必须严格是 T* T::Clone() const。

(3) 某程序员想写一个模板，此模板可以要求（或者可以检测出）：它在实例化时所使用的类型具有一个 Clone() 成员函数。这个程序员采用的方案基于这样一个要求：提供 Clone() 的类必须派生于某个已有的 Cloneable 基类。演示如何写出下面的模板：

```
template<typename T>
class X
{
    // ...
};
```

(a) 此模板要求 T 派生于 Cloneable；
 (b) 如果 T 派生于 Cloneable，此模板提供一个可供选择的实现；否则，此模板工作于某种缺省模式。

(4) 为了要求（检测）Clone() 存在，上面第 3 点所采用的方法是最佳方案吗？分析其它可能的方案。

(5) 知道模板的参数类型 T 派生于某个其它类型，这对模板来说有什么用处呢？知道这种派生关系能带来某种好处吗？而且，这种好处在没有继承关系的情况下就无法获得吗？

解答

1. 什么是 traits 类？

引用 C++ 标准 [C++98] 第 17.1.18 节的定义，一个 traits 类（特征类）是：

一个封装了一组类型 (*types*) 与函数 (*functions*) 的类，以使模板类与模板函数可以操纵实例化类型的对象。^②（译注：“实例化类型”即模板实例化时使用的类型。为了中文表述上的方便，故简称之为。）

基本概念是：traits 类是模板实例，用于携带 traits 模板被实例化时类型的额外信息——特别是，可以被其它模板使用的信息。它所带来的好处是：在对某个类 C 不做任何修改的情况下，`T<C>`这个 traits 类能让我们记录 C 的（以上所说的那种）额外信息。

例如，标准中有一个 `std::char_traits<T>`，它提供 T 这种字符似 (character-like) 类型的有关信息——特别是，如何对这种 T 对象进行比较和操作的有关信息。这些信息用在如 `std::basic_string` 和 `std::basic_ostream` 之类的模板中，使得这些模板可以和字符类型工作，而这些字符类型不必是 `char` 或 `wchar_t`；这些模板也可以和用户自定义类型工作——如果你为这种类型提供了一个合适的 `std::char_traits` 特殊化。类似地，`std::iterator_traits` 提供了迭代器的有关信息，其它模板，特别是算法和容器，可以很好地使用这些信息。甚至 `std::numeric_limits` 也加入了 traits 的行列，对于那些在特定平台和编译器上实现的各种数字类型，`std::numeric_limits` 提供了这些数字类型的处理能力和行为特征的有关信息。

更多的例子请参见：

- 条款 30 和 31 中关于智能指针成员的讨论。
- *Exceptional C++* [Sutter00] 的条款 2 和条款 3。这两个条款演示了如何定制 `std::char_traits`，从而最终定制 `std::basic_string` 的行为。
- C++ Report 2000 年 4 月、5 月及 6 月号，其中包含有关 traits 的多个精彩专栏。

^② 这里，封装 (encapsulate) 的意思是指将所有东西集中在一个地方，而不是指隐藏在什么外壳之中。traits 类中的一切通常都是公有的，而且，一般来说它们确实都是通过 struct 模板来实现的。

对成员函数的需求

2. 示范如何检测和运用模板参数的成员，请使用下面这个具有启发性的例子：假设你想写一个类模板 C，能够实例化此模板的类型必须具有一个名为 Clone() 的 const 成员函数，此函数不带参数，返回值为指针，指向同种类型的对象。

```
// 例 4-2
//
// T 必须提供 T* T::Clone() const
template<typename T>
class C
{
    // ...
};
```

注意到一条很显然的事实：如果在模板 C 中写一段代码，让它去调用不带参数的 T::Clone()，那么，如果实际上没有这样一个“不带参数就可以被调用”的 T::Clone()，这段代码就不能够通过编译。

有个例子可以用来说明后面那条注意事项，请看下面的代码：

```
// 例 4-2(a)：初次尝试，某种程度上体现了对 Clone() 的需要
//
// T 必须提供 /*...*/ T::Clone( /*...*/ )
template<typename T>
class C
{
public:
    void SomeFunc( const T* t )
    {
        // ...
        t->Clone();
        // ...
    }
};
```

例 4-2(a)存在的第一个问题是：它可能根本就没有提出任何需求。在一个模板中，只有被实际使用了的成员函数才会被实例化。^③ 如果 SomeFunc() 永远没有被使用，它就永远不会被实例化；这样，即使类型 T 没有任何类似 Clone() 这样的东西，C 却可以很容易地被实例化。

解决之道是：将“体现需求”的代码置入某个一定会被实例化的函数中。很多人把它放到构造函数中，因为只要使用 C，就不可能不在什么地方调用它的构造函数，不是吗？（例如，[Stroustrup94]中提到过这种方法）很正确！但构造函数可能有多个。那么，为了安全起见，我们就得把“体现需求”的代码放到每一个构造函数中。

^③ 最终，所有的编译器将都遵循这条规则。你目前使用的编译器可能还是实例化所有的函数，而不只是那些被使用了的函数。

一个更简单的方法是把它放到析构函数中。毕竟，析构函数只有一个；而且，使用类 C 但不调用它的析构函数的情况是不大可能的（动态创建了 C 的对象而从来没有销毁它）。所以，或许，在析构函数里放置“体现需求”的代码是稍微简单一些的方案：

```
// 例 4-2(b)：修正方案，体现了对 Clone() 的需求
//
// T 必须提供 /*...*/ T::Clone( /*...*/ )
template<typename T>
class C
{
public:
    ~C()
    {
        // ...
        T t; // 某种浪费，还需要 T 有一个缺省构造函数
        t.Clone();
        // ...
    }
};
```

这一方案还是不能完全令人满意。我们暂时撇开它不管，在多做一些分析后，我们会很快回过头来，进一步完善这一方案。

这样，我们就面临第二个问题：例 4-2(a) 和例 4-2(b) 都只是依赖于约束条件，而没有测试这些约束条件。（例 4-2(b) 的情况更严重，因为它的做法很浪费，仅仅为了限定一个约束条件，却没必要地增加了运行时期的代码。）

但这一事实并不足以回答本问题。因为，在调用不带参数的 T::Clone() 时，如果存在一个带缺省参数并且（或者）不返回 T* 的 Clone()，调用也会成功。

如果类型 T 有一个看起来像 T* T::Clone() 那样的函数，例 4-2(a) 和例 4-2(b) 中的代码确实会非常顺利地通过编译。问题在于，如果有 void T::Clone()，或 T* T::Clone(int = 42)，或其它什么具有古怪原型（signature）的函数，例如 T* T::Clone(const char* = "xyzzy")，只要它们不需要参数就可以被调用，那么，例 4-2(a) 和例 4-2(b) 中的代码也会很容易地通过编译。（关于这一点再补充一句：即使完全没有 Clone() 成员函数，它也可能会通过编译——只要写一个宏，把 Clone 这个名字改变为其它什么东西。对于这样的情况，我们几乎无能为力。）

尽管这些做法用在某些程序里可能不错，但它不是我们提出问题的本意所在。我们的需求更严格：

这里，我们的目标明确限定为：T 提供的函数必须严格是 T* T::Clone() const。

所以我们有了下面的做法：

```
// 例 4-2(c)：更好的方案，严格要求
// 是 T* T::Clone() const
//
// T 必须提供 T* T::Clone() const
template<typename T>
class C
{
public:
    // 放在 C 的析构函数中(比放在
    // 每一个构造函数中要简单)：
    ~C()
    {
        T* (T::*test)() const = &T::Clone;
        test; // 禁止掉未使用的变量所造成的警告
        // 这种未使用的变量可能会被编
        // 译器的优化程序完全消除掉

        // ...
    }

    // ...
};
```

或者，使代码更清晰并更具扩充性一点：

```
// 例 4-2(d)：另一选择方案，严格要求
// 是 T* T::Clone() const
//
// T 必须提供 T* T::Clone() const
template<typename T>
class C
{
    bool ValidateRequirements() const
    {
        T* (T::*test)() const = &T::Clone;
        test; // 禁止掉未使用的变量所造成的警告
        // ...
        return true;
    }

public:
    // 放在 C 的析构函数中(比放在
    // 每一个构造函数中要简单)：
    ~C()
    {
        assert( ValidateRequirements() );
    }

    // ...
};
```

有了函数 `ValidateRequirements()`，代码更具可扩充性，因为它为我们提供了一个非常干净的地方，便于将来增加任何“需求检查”功能。函数调用是在 `assert()` 中进行的，这进一步保证：针对“需求检查”的所有跟踪代码不会出现在发行版本中。

约束类（Constraints Classes）

但还有更干净的方法。下面的技术由 Bjarne Stroustrup 在他的 *C++ Style and Technique FAQ* [StroustrupFAQ] 一书中提出；当然，还得感谢 Alex Stepanov 和 Jeremy Siek，他们提出了对函数指针的这种使用方法。^④

假设我们写了下面这个约束类 `HasClone`：

```
// 例 4-2(e)：使用约束继承 (constraints inheritance),
// 要求准确无误的 T* T::Clone() const
//
// HasClone 要求 T 必须提供
// T* T::Clone() const
template<typename T>
class HasClone
{
public:
    static void Constraints()
    {
        T* (T::*test)() const = &T::Clone;
        test; // 禁止掉未使用的变量所造成的警告
    }
    HasClone() { void (*p)() = Constraints; }
};

```

于是，我们就有了一个非常“优雅”——我敢说“酷”吗？——的方法，以在编译时期实施约束：

```
template<typename T>
class C : HasClone<T>
{
    // ...
};
```

这个方案的设计思想很简单：每个 `C` 的构造函数必然调用 `HasClone<T>` 的缺省构造函数，此构造函数什么也没做，仅做约束检查。如果约束检查失败，大多数编译器会产生一条可读性极强的出错信息。`HasClone<T>` 的派生关系就相当于一条断言（assertion），它以一种易于诊断的方式检查出 `T` 的某一特性。

^④ 参见 http://www.gotw.ca/publications/mxc++/bs_constraints.htm

对继承的需求，方式 1：

IsDerivedFrom1 值辅助类 (Value Helper)

3. 某程序员想写一个模板，此模板可以要求（或者可以检测出）：它在实例化时所使用的类型具有一个 `Clone()` 成员函数。这个程序员采用的方案基于这样一个要求：提供 `Clone()` 的类必须派生于某个已有的 `Cloneable` 基类。演示如何写出下面的模板：

```
template<typename T>
class X
{
    // ...
};
```

- a) 此模板要求 T 派生于 `Cloneable`。

我们将看到两种方案，二者都可行。第一种方案需要一点技巧，稍微复杂一些；第二种方案则简单精巧。两种方案都值得在此探讨，因为二者都展示了一些很有趣的设计技术；即使其中只有某项技术用在这儿刚好更合适一些，但对这些技术都了解一下还是很有好处的。

第一种方案基于 Andrei Alexandrescu 在“*Mappings Between Types and Values*”[Alexandrescu00a]一文中提出的思想。首先，我们定义一个辅助模板，用来测试候选类型 D 是否派生于 B。测试的方法是，检查一个指向 D 的指针能否被转换为一个指向 B 的指针。下面就是一种做法，和 Alexandrescu 的方法类似：

```
// 例 4-3(a): IsDerivedFrom1 求值辅助类
//
// 优点：可用于编译时期对值的检查
// 缺点：太复杂
//
template<typename D, typename B>
class IsDerivedFrom1
{
    class No { };
    class Yes { No no[2]; };

    static Yes Test( B* ); // 声明但未定义
    static No Test( ... ); // 声明但未定义

public:
    enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
};
```

看明白了吗？继续往下阅读之前先琢磨一下这段代码。

* * * * *

上面的花招倚仗以下三点：

- (1) Yes 和 No 的大小 (size) 不同。为了保证这一点，我们让 Yes 包含一个数组，此数组保存的是 No 对象，且 No 对象的数目大于 1。（是的，双重否定往往变成肯定。但在此处，实际上 No 不是 no. ②）
- (2) 重载函数的解析和 sizeof 值的确定是在编译时期完成的，而非运行时期。
- (3) Enum 的值在编译时期得出，并可以在编译时期使用。

让我们对 enum 的定义进行更详细的分析。首先，看看最里层的部分：

```
Test(static_cast<D*>(0))
```

它只是涉及到一个名为 Test 的函数，并假装传给这个函数一个 D*——本例中，我们通过“对空指针进行合适的类型转换”来实现这一点。注意，这里其实什么也没有做，没有代码被生成，所以指针绝没有被解引用 (dereferenced)。或者说，基于上面的原因，指针甚至根本就没有被真正地创建。我们所做的一切是创建了一个类型化的表达式 (typed expression)。现在，编译器知道 D 为何物，在编译期间进行重载解析时，编译器会根据 D 的类型来决定该选择 Test() 的哪一个重载函数：如果 D* 可以被转换为 B*，就选择 Test(B*)——返回值为 Yes 的那一个；否则，选择 Test(...)——返回值为 No 的那个。

那么，很明显，下一步是去检查哪个重载函数会被选择：

```
sizeof(Test(static_cast<D*>(0))) == sizeof(Yes)
```

这个表达式的求值也是完全在编译时期进行的，如果 D* 可以被转换为 B*，结果为 1；否则为 0。这恰好是我们想知道的一切，因为，当且仅当 D 从 B 派生，或者 D 就是 B 的时候，D* 才能被转换为 B*。^③

既然已经计算出需要知道的东西，下一步只需要在某处存储其结果。这里所说的“某处”必须是这样一个地方：在编译时期，它能够被设置 (set)，值还能够被使用。很幸运，enum 刚好可以担当这个角色：

```
enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
```

在这个例子中，对 Cloneable 来说，我们不必在意 D 和 B 是不是相同的类型。我们只想知道，D 是否能够被多态地作为 B 来使用，这正是 IsDerivedFrom1 所检测的内容。B 当然能够作为 B 使用，这一点再浅显不过。

仅此而已。现在，我们可以利用这些成果构造出问题的答案，即：

```
// 例 4-3(a)，继续：使用 IsDerivedFrom1 辅助类
// 实施“从 Cloneable 派生”的约束
//
```

^③ 或者 B 刚好是 void.

```

template<typename T>
class X
{
    bool ValidateRequirements() const
    {
        // 需要一个 typedef, 否则, 对 assert 来说,
        // “,”会被解释为宏(macro)的参数分隔符
        typedef IsDerivedFrom1<T, Cloneable> Y;

        // 运行时检查, 但不需要多少工作量就
        // 可以很容易地将其转化为编译时检查
        assert( Y::Is);

        return true;
    }

public:
    // 放在 X 的析构函数内(比放在
    // X 的每个构造函数内要简单):
    ~X()
    {
        assert( ValidateRequirements() );
    }

    // ...
};

```

对继承的需求，方式 2：

IsDerivedFrom2 约束基类 (Constraints Base Class)

至此你可能已经注意到，采用 Stroustrup 的方法，我们可以写出一个功能上等价的版本，但句法上更漂亮：

```

// 例 4-3(b): IsDerivedFrom2 约束基类
//
// 优点: 编译时期求值, 易于直接使用
//
// 缺点: 不能直接应用于编译时期值的测试
//
template<typename D, typename B>
class IsDerivedFrom2
{
    static void Constraints(D* p)
    {
        B* pb = p;
        pb = p; // 禁止掉未使用的变量所造成的警告
    }

protected:
    IsDerivedFrom2() { void(*p)(D*) = Constraints; }
};

```

```
// B 为 void 的情况下，强制失败
template<typename D>
class IsDerivedFrom2<D, void>
{
    IsDerivedFrom2() { char* p = (int*)0; /* 错误 */ }
};
```

现在，检测更简单了：

```
// 例 4-3(b)，继续：使用 IsDerivedFrom2 约束基类
// 实施“从 Cloneable 派生”的约束
//
template<typename T>
class X : IsDerivedFrom2<T, Cloneable>
{
    // ...
};
```

对继承的需求，方式 3：

结合以上两种方案的 IsDerivedFrom

较之 IsDerivedFrom2，IsDerivedFrom1 的主要优点在于：IsDerivedFrom1 提供了一个 enum 值，这个值在编译时期产生，还可以在编译时期检测。对这里的类 X 示例来说，这一点不重要；但在下一节中，这一点很重要，因为我们需要通过切换这样一个值来在编译时期选择不同的 traits 实现。另一方面，通常情况下，IsDerivedFrom2 在使用上具有很大的方便性，当我们需要在模板参数上实施某种约束，以保证某种设施的存在时，我们不需要做什么特别的事，譬如，不需要在不同实现间进行选择。我们可以同时提供以上两种版本，但功能上的重复和相近是个问题；还有，命名也是个问题。想区分这两个版本，我们没办法比前面做得更好了——即，在名称上添加点什么多余的东西以示区别；这样一来，用户就得时时记住他们想要的是 IsDerivedFrom1 还是 IsDerivedFrom2。这太丑陋了。

为什么不能做到鱼和熊掌兼得呢？我们不妨将这两种方案结合起来：

```
// 例 4-3(c)：IsDerivedFrom 约束基类，
// 并具有测试值
//
template<typename D, typename B>
class IsDerivedFrom
{
    class No {};
    class Yes { No no[2]; };

    static Yes Test( B* ); // 未定义
    static No Test( ... ); // 未定义
```

```

static void Constraints(D* p) { B* pb = p; pb = p; }

public:
    enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };

    IsDerivedFrom() { void(*p)(D*) = Constraints; }
};

```

选择不同的实现

例 4-3(a) 中的方案很不错，也很完整，它要求 T 必须是一个 Cloneable。但如果 T 不是 Cloneable 怎么办？如果我们需要执行某种其它行为又该怎么办？也许我们应该让事情更灵活一些——这给我们带来了问题的第二部分。

b) [...] 如果 T 派生于 Cloneable，此模板提供一个可供选择的实现；否则，此模板工作于某种缺省模式

著名的“额外一层间接”（extra level of indirection）可以解决很多计算问题，我们也借助它来完成这里的任务。在这儿，额外一层间接是以辅助模板的形式出现的：X 将使用例 4-3(c) 中的 IsDerivedFrom，并使用这个辅助模板的部分特殊化（partial specialization）来完成在“Cloneable”实现和“非 Cloneable”实现间的切换。（注意，这要求 IsDerivedFrom1 具有“在编译时期可以检测的值”，这个值也应该并入到 IsDerivedFrom 中，这样我们才有可供检测的东西，从而做到在不同的实现间切换。）

```

// 例 4-3(d): 如果从 Cloneable 派
// 生，用 IsDerivedFrom 来处理;
// 否则，做其它什么事
//
template<typename T, int>
class XImpl
{
    // 一般情况: T 没有从 Cloneable 派生
};

template<typename T>
class XImpl<T, 1>
{
    // T 从 Cloneable 派生
};

template<typename T>
class X
{
    XImpl<T, IsDerivedFrom<T, Cloneable>::Is> impl_;
    // ...委托给 impl_ ...
};

```

明白这段代码是如何运作的吗？让我们用一个例子很快地将其工作流程走一遍：

```
class MyCloneable : public Cloneable { /* ... */ };
X<MyCloneable> x1;
```

X<T>中 `impl_` 的类型是：

```
XImpl<T, IsDerivedFrom<T, Cloneable>::Is>
```

本例中，T 为 `MyCloneable`，所以 X<MyCloneable> 中 `impl_` 的类型是：

```
XImpl<MyCloneable, IsDerivedFrom<MyCloneable, Cloneable>::Is>
```

求值后，它实际上是：

```
XImpl<MyCloneable, 1>
```

它利用了 `MyCloneable` 从 `Cloneable` 派生这一事实，从而使用了 `XImpl` 特殊化版本。

但如果我们用某个其它类型来实例化 X，又会怎样呢？请看：

```
X<int> x2;
```

现在 T 为 `int`，所以 X<int> 中 `impl_` 的类型是：

```
XImpl<int, IsDerivedFrom<int, Cloneable>::Is>
```

求值后，它实际上是：

```
XImpl<int, 0>
```

这次使用的是非特殊化的 `XImpl`。干得很漂亮，不是吗？一旦写完这些代码，它一点都不难使用。从用户的角度来看，复杂性被隐藏在 X 的内部。从 X 的设计者的角度来看，他只是直接复用（reuse）已经封装在 `IsDerivedFrom` 中的功能，而不必了解魔法是如何运作的。

注意，我们不是在扩散（proliferate）模板实例。对任何一个 T，总是只有一个 `XImpl<T, ...>` 会被实例化，要么是 `XImpl<T, 0>`，要么是 `XImpl<T, 1>`。虽然 `XImpl` 的第二个参数理论上可以取任何整数值，但我们在这儿已经作了处理，使得整数值只会是 0 或 1。（既然如此，为什么不用 `bool` 而用 `int` 呢？答案是，为了扩充性：使用 `int` 没有坏处，相反，这样做的话，将来如果有需要，我们可以很容易地增加其它不同的实现——例如，假设我们以后又想对另一种继承层次结构提供支持，而这个继承层次结构具有一个和 `Cloneable` 相似但带有不同接口的基类）

需求（Requirements）与 Traits

4. 为了要求（检测）`Clone()` 存在，上面第 3 点所采用的方法是最佳方案吗？分析其它可能的方案。

问题 3 中的方法很俏皮，但很多情况下我更倾向于使用 traits——它们往往很简单（有一种情况除外：对一个继承层次结构中的每个类，它们都必须进行特殊化的时候），而且如条款 30 和 31 所示，它们更具可扩充性。

基本想法是创建一个 traits 模板，在当前情况下，此 traits 模板的唯一任务是实现 `Clone()` 操作。这个 traits 模板看起来很像 `XImpl`，它有一个通用的非特殊化的版本，用来做一些通用的事情，还可能有多个特殊化的版本，用来处理某些类，这些类可能提供的是性能更好的 `clone` 操作，或者仅仅是不同方式的 `clone` 操作。

```
// 例 4-4: 使用 traits 而非 IsDerivedFrom 来使用
// 可能存在的 Clone 功能；若不存在，则做别的事情。
// 需要为每个 Cloneable 类提供一个特殊化。
//
template<typename T>
class XTraits
{
public:
    // 一般情况：使用拷贝构造函数
    static T* Clone( const T* p ) { return new T( *p ); }

};

template<>
class XTraits<MyCloneable>
{
public:
    // MyCloneable 从 Cloneable 派生，所以使用 Clone()
    static MyCloneable* Clone( const MyCloneable* p )
    {
        return p->Clone();
    }
};

// ...其它，针对从 Cloneable 派生的每一个类
```

于是，`X<T>`会调用合适的 `XTraits<T>::Clone()`，执行正确的操作。

`Traits` 和例 4-3(b) 中那个普通的 `XImpl` 之间的区别主要在于：有了 `traits`，当用户定义了某个新类型的时候，如果想将这个新类型用于 `X`，所要做的大部分工作都在 `X` 的外部——我们只用特殊化 `traits` 模板，为新类型“做正确的事”就可以了。这就比前面问题 3 中那个相对死板的方案更具可扩充性，因为后者是在 `XImpl` 的实现中完成所有的选择工作，没有为了提高可扩充性而将其开放。`Traits` 还允许使用其它 `clone` 方法（method），而不仅仅是特别命名为 `Clone()` 的某个函数——而且这个函数还得从一个特别命名的基类继承而来；这也提供了另外的灵活性。

更详细的介绍，请参见条款 31 的例 31-2(d) 和 31-2(e)，那儿有一个非常相似的 `traits` 实现的示例，但代码更长。

注意：上面所说的 `traits` 方案的主要缺点是：针对继承层次结构中的每一个类，它都需要单独的特殊化。有办法可以一次性地为整个层次结构中的类提供 `traits`，从而不必枯燥地写出大量的特殊化。请参见 [Alexandrescu00b]，那里介绍了一种很漂亮的

技术来完成这一工作。使用那一技术，你就不用对外部类层次结构中的基类（譬如本例中的 `Cloneable` 基类）伤筋动骨。

继承与 Traits

5. 知道模板的参数类型 `T` 派生于某个其它类型，这对模板来说有什么用处呢？知道这种派生关系能带来某种好处吗？而且，这种好处在没有继承关系的情况下就无法获得吗？

对一个模板来说，就算知道它的一个模板参数从某个给定的基类继承，这也不能让它获得“使用 traits 无法获得”的额外好处。使用 traits 仅有的一一个真正的缺点是，在一个庞大的继承体系中，为了处理大量的类，需要写大量的特殊化代码；不过，运用某些技术可以减轻或者消除这一缺点。

本条款的主要目的在于说明：与某些人的想法相反，“为了处理模板中的分类（categorization）而使用继承”不足以成为使用继承的理由。traits 提供了更通用的机制；当用一个新类型——例如来自第三方程序库中的某个类型——来实例化一个现有模板的时候，此类型可能很难从某个预先确定的基类派生，此时，traits 体现了更强的可扩充性。

条款 5: typename

难度：7

“(类型)名称之中有什么玄妙之处？”在这个练习中，我们将运用标准库中一种常见的手法，向大家展示为什么要使用 `typename`，以及如何使用。

- (1) 什么是 `typename`？它有什么用处？
- (2) 下面的代码有错吗？如果有，错在哪儿？

```
template<typename T>
class X_base
{
public:
    typedef T instantiated_type;
};

template<typename A, typename B>
class X : public X_base<B>
{
public:
    bool operator()( const instantiated_type& i ) const
    {
        return i != instantiated_type();
    }

    // ... 更多代码 ...
};
```

解答**1. 什么是 typename? 它有什么用处?**

这将我们带回到名称查找 (name lookup) 的话题上。一个启发式的例子是：模板中那种具有依赖性的名称——即，假如有下面这样的代码：

```
// 例 5-1
//
template<typename T>
void f()
{
    T::A* pa; // 这行代码有什么用?
}
```

`T::A` 是一个有依赖性的名称 (dependent name)，因为它依赖于模板参数 `T`。在这个例子中，程序员可能希望 `T::A` 是 `T` 中的嵌套类或 `typedef`，因为如果是其它什么东西，例如静态成员变量或函数，则这段代码就无法通过编译。这里有一个问题，因为 C++ 标准中有这样的叙述：

如果一个名称被使用在模板声明或定义中并且依赖于模板参数，则这个名称不被认为是一个类型的名称，除非名称找到了一个合适的类型名称，或者这个名称用关键字 `typename` 修饰。

这将我们带到主要问题前：

2. 下面的代码有错吗？如果有，错在哪儿？

本例想说明的问题是：为什么要使用 `typename` 来引用 (refer to) 有依赖性的名称，以及如何使用，从而启发我们回答这个问题：“名称之中有什么玄妙之处？”

```
// 例 5-2
//
template<typename T>
class X_base
{
public:
    typedef T instantiated_type;
};

template<typename A, typename B>
class X : public X_base<B>
{
public:
    bool operator()( const instantiated_type& i ) const
{
```

```

        return i != instantiated_type();
    }
    // ... 更多代码 ...
};

```

对有依赖性的名称使用 typename

X 的问题在于，“instantiated_type”本来想引用的是从基类 X_base继承而来的 typedef。不幸的是，在编译器解析 X<A, B>::operator()() 的内联定义之时，有依赖性的名称（即，依赖于模板参数的名称，例如继承而来的 X_Base::instantiated_type）是不可见的，所以编译器将会发出抱怨，因为它不知道 instantiated_type 指的是什么。有依赖性的名称只会在后来可见，也就是模板被真正实例化的时候。

如果你奇怪编译器为什么没有办法解决这个问题，那么，假设你自己是编译器，问问自己该如何猜出 instantiated_type 在此处的含义。这是一个很有趣的练习。但最终结果一定是：你猜不出来，因为你还不知道 B 为何物，你也不知道随后是否会有某种 X_base 的特殊化，使得 X_base::instantiated_type 成为意想不到的什么东西——任何类型名称，或者，甚至是个成员变量。在上面未被特殊化的 X_base 模板中，X_base<T>::instantiated_type 总会是 T，但在特殊化的时候，我们无法阻止有人去改变它的含义，例如：

```

template<>
class X_base<int>
{
public:
    typedef Y instantiated_type;
};

```

是的，虽然这样做会导致 typedef 的名称有点让人误解，但它是合法的。或者，有人会这样做：

```

template<>
class X_base<double>
{
public:
    double instantiated_type;
};

```

现在，名称不那么让人误解，但模板 X 不能将 X_base<double> 作为基类，因为 instantiated_type 是一个成员变量，不是一个类型名称。

总而言之，编译器不可能知道如何解析 X<A, B>::operator()() 的定义，除非我们告诉它 instantiated_type 是什么——至少，得告诉它是类型还是其它什么东西。这里，我们希望它是个类型。

要想告诉编译器这种东西是类型名称，方法是为它加上 `typename` 关键字。这里我们有两种做法。不那么优雅的做法是，简单地在所有用到了 `instantiated_type` 的地方都添上 `typename`:

```
// 例 5-2(a): 有点恐怖
//
template<typename A, typename B>
class X : public X_base<B>
{
public:
    bool operator()(
        const typename X_base<B>::instantiated_type& i
    ) const
    {
        return i != typename X_base<B>::instantiated_type();
    }

    // ... 更多代码 ...
};
```

我相信，看到这种做法你会敬而远之。同样，在这种场合下，`typedef` 的出现将会让代码的可读性更好，我们只用提供另一个 `typedef`，函数定义中的其余部分就会像最初所写的那样：

```
// 例 5-2(b): 好一些
//
template<typename A, typename B>
class X : public X_base<B>
{
public:
    typedef typename X_base<B>::instantiated_type
        instantiated_type;

    bool operator()( const instantiated_type& i ) const
    {
        return i != instantiated_type();
    }

    // ... 更多代码 ...
};
```

继续往下阅读之前，问问自己：增加这样一个 `typedef` 是不是让你觉得有点怪怪的？

次要（微妙）的一点

我本可以用更简单的例子来说明问题（一些例子可以在 C++ 标准第 14.6/2 节中找到），但那不足以指出其奇怪之处：空基类 `X_base` 存在的全部理由在于提供 `typedef`。但是，派生类往往最终又一次对它使用了 `typedef`。

不觉得有点重复吗？是的，但只是一点儿。毕竟，它还是 `X_base` 的特殊化，`X_base` 会负责确定合适的类型应该是什么，对于不同的特殊化，这个类型会变化。

标准库中包含许多这样的基类，即所谓“充斥着 `typedef`”的基类，它们就得以这样的方式来使用。我希望，本条款有助于澄清“派生类为什么要看似多余地对 `typedef` 再一次进行类型定义（re-`typedef`）”的疑问，并且证明：这种情况并不是语言设计中的一个缺陷，它只不过反映了下面这个古老问题的另一面：

“名称之中有什么玄妙之处？”

附笔

额外地，这里奉献的是有关 `typename` 的一则幽默，用代码写成：

```
#include <iostream>

class Rose {};

class A { public: typedef Rose rose; };

template<typename T>
class B : public T { public: typedef typename T::rose foo; };

template<typename T>
void smell( T ) { std::cout << "awful"; }

void smell( Rose ) { std::cout << "sweet"; }

int main()
{
    smell( A::rose() );
    smell( B<A>::foo() ); // :-
}
```

条款 6：容器、指针和“不是容器的容器”

难度：5

油和水的确不能溶合。指针和标准容器可以做得更好吗？

1. 请看下面的代码：

```
vector<char> v;
```

```
// ... 填充 v ...
vector<char>::iterator i = v.begin();
// ... 使用*i ...
```

通常，当你想指向一个容器内部的对象时，一条不错的准则是：尽量使用迭代器而不使用指针。毕竟，迭代器和指针往往是在同样的情况下以同样的方式失效。迭代器存在的一个理由是，它提供了一种方式，用以“指向”一个被包含对象。如果可以选择，尽量使用迭代器来指向容器内部。

不幸的是，使用指向容器内部的指针所得到的效果，并不总能通过迭代器来得到。使用迭代器有两个潜在的缺陷，只要其中一个落到你头上，你就要继续使用指针。

- (1) 能使用指针的地方，不一定总能方便地使用迭代器。（见下面的例子）。
- (2) 如果迭代器是一个对象而不是一个普通指针（bald pointer），使用迭代器会招致空间和性能上的额外开销。

例如，假设有一个 `map<Name,PhoneNumber>`，它在程序启动的时候载入，在此之后仅用于查询。也就是说，给出一个名字，就可以很容易地在这个已有的数据字典中查找到相应的电话号码。但如果需要进行反向查找怎么办？一个干净的手法是，再构造第二个数据结构，例如 `map<PhoneNumber*,Name*,Deref>`，它让你可以进行反向查找，但避免了双倍的存储开销；因为，使用指针，就不需要将每个名字和电话号码存储两次。第二个结构拥有的只是指针，指向第一个结构的内部。

这个方案很不错，可以有效地工作；但请注意，如果不使用指针而是使用迭代器，就很难达到这种效果。为什么？因为，作为当然的候选者，`map<Name,PhoneNumber>::iterator` 指向的是 `pair<const Name, PhoneNumber>`，没有某种方便的方法可以得到一个迭代器，让它单独指向姓名部分或电话号码部分。（我们可以存储整个迭代器，并在每个地方显式地写上`->first` 和 `->second`；但这样一来，编写代码就很不方便，而且这也意味着，那个支持反向查找的 `map` 必须重新设计，或者应该换成别的数据结构。）

这将我们带到本条款的下一个（在我看来最有趣的）问题面前。

2. 现在分析下面的代码：

```
// 例 6-2：这段代码合法吗？
//
template<typename T>
void f( T& t )
{
    typename T::value_type* p1 = &t[0];
    typename T::value_type* p2 = &*t.begin();
```

为什么要获取指向容器内部的指针或引用？

1. 请看下面的代码：

```
// 例 6-1(a): 这段代码合法吗? 安全吗? 好吗?  
//  
vector<char> v;  
  
// ... 填充 v ...  
  
char* p = &v[0];  
  
// ... 使用*p ...
```

这段代码合法吗？

C++ 标准保证：如果一个符合标准的序列（例如 `vector<char>`）提供了 `operator[]()` 运算符，这个运算符必然返回类型 `char` 的左值（lvalue），而左值是可以取得地址的⁽⁶⁾（详注：参阅[Stroustrup00]第 4.9.6 及 11.2.2 节）。所以这个问题的答案是：是的，只要 `v` 非空，例 6-1(a) 中的代码是完全合法的。

这段代码安全吗？对于这种获取指向容器内部的指针的做法，一些程序员第一次看到时会大吃一惊，但这里的答案是：是的，它是安全的，只要我们清楚地知道指针什么时候会失效，也就是，等价的迭代器什么时候会失效。例如，如果我们开始在这个容器中插入或删除元素，很明显，由于底部内存发生移动，不仅迭代器会失效，任何指向容器内部的指针都会失效。

但是，这样做有意义吗？再一次，答案是肯定的：拥有一个指向容器内部的指针或引用绝对有意义。一个常见的情况是：在程序启动时，你将一个数据结构读入内存，然后一直不去修改它，但需要以不同的方式对它进行访问。在这种情况下，如果有另外的数据结构包含指向主容器内部的指针，从而对不同的访问方式进行优化，这就很有意义了。我将在下一节给出一个简单的例子。

如何改进这段代码

无论合法与否，如何改进它？

例 6-1(a) 可以通过下面的方法改进：

```
// 例 6-1(b): 一种改进 (在可能的情况下)  
//  
vector<char> v;
```

⁽⁶⁾ 在本条款中，我用不同的方式来表达这个术语——例如，“指向容器内部的指针”、“指向容器内部对象的指针”、“指向被包含对象的指针”——它们指的都是一回事。

```
// ... 填充 v ...
vector<char>::iterator i = v.begin();
// ... 使用*i ...
```

通常，当你想指向一个容器内部的对象时，一条不错的准则是：尽量使用迭代器而不使用指针。毕竟，迭代器和指针往往是在同样的情况下以同样的方式失效。迭代器存在的一个理由是，它提供了一种方式，用以“指向”一个被包含对象。如果可以选择，尽量使用迭代器来指向容器内部。

不幸的是，使用指向容器内部的指针所得到的效果，并不总能通过迭代器来得到。使用迭代器有两个潜在的缺陷，只要其中一个落到你头上，你就要继续使用指针。

- (1) 能使用指针的地方，不一定总能方便地使用迭代器。（见下面的例子）。
- (2) 如果迭代器是一个对象而不是一个普通指针（bald pointer），使用迭代器会招致空间和性能上的额外开销。

例如，假设有一个 `map<Name,PhoneNumber>`，它在程序启动的时候载入，在此之后仅用于查询。也就是说，给出一个名字，就可以很容易地在这个已有的数据字典中查找到相应的电话号码。但如果需要进行反向查找怎么办？一个干净的手法是，再构造第二个数据结构，例如 `map<PhoneNumber*,Name*,Deref>`，它让你可以进行反向查找，但避免了双倍的存储开销；因为，使用指针，就不需要将每个名字和电话号码存储两次。第二个结构拥有的只是指针，指向第一个结构的内部。

这个方案很不错，可以有效地工作；但请注意，如果不使用指针而是使用迭代器，就很难达到这种效果。为什么？因为，作为当然的候选者，`map<Name,PhoneNumber>::iterator` 指向的是 `pair<const Name, PhoneNumber>`，没有某种方便的方法可以得到一个迭代器，让它单独指向姓名部分或电话号码部分。（我们可以存储整个迭代器，并在每个地方显式地写上`->first` 和 `->second`；但这样一来，编写代码就很不方便，而且这也意味着，那个支持反向查找的 `map` 必须重新设计，或者应该换成别的数据结构。）

这将我们带到本条款的下一个（在我看来最有趣的）问题面前。

2. 现在分析下面的代码：

```
// 例 6-2：这段代码合法吗？
//
template<typename T>
void f( T& t )
{
    typename T::value_type* p1 = &t[0];
    typename T::value_type* p2 = &t.begin();
```

```
// ... 使用*p1 和*p2 ...
}
```

继续往下阅读之前，考虑以下问题：这段代码合法吗？如果合法，在什么条件下合法？特别是，要让这段代码合法，对 T 有什么要求？（不考虑运行时期的情况，例如 t 是否刚好处于合适的状态，使得 t[0] 可以被调用等等。在这里，我们感兴趣的是程序设计的合法性）

何时一个容器并非容器？

那好，例 6-2 合法吗？简而言之，是的，它“可以”是合法的。

要想得到更详细的答案，就得思考一下，怎样的 T 会使代码合法：一个合适的 T 必须具有怎样的特征和能力？让我们来做一些试探性的分析：

(a) 要想让表达式 `&t[0]` 合法，`T::operator[](0)` 必须存在，而且返回值必须能够被 `operator&()` 操作；而 `operator&()` 必须返回一个合法的 `T::value_type*`（或者其它什么东西，只要它可以有意义地转换为合法的 `T::value_type*`）。

具体来说，如果容器满足 C++ 标准中容器和序列的条件，并且对于可选的 `operator[](0)`，它也提供了实现（因为我们必须让这个运算符返回一个指向内部对象的引用），这个容器就符合上面的要求。根据定义，于是你可以取得被包含对象的地址。

(b) 要想让表达式 `&t.begin()` 合法，`T::begin()` 必须存在，而且返回值必须能够被 `operator*()` 操作；而 `operator*()` 的返回值也必须能够被 `operator&()` 操作；同时，`operator&()` 必须返回一个合法的 `T::value_type*`（或者其它什么东西，只要它可以有意义地转换为合法的 `T::value_type*`）。

具体来说，只要容器的迭代器满足 C++ 标准中迭代器的条件（因为 `begin()` 返回的迭代器通过 `operator*()` 解引用（dereferenced）后，必须返回一个指向被包含对象的引用），这些容器就符合以上要求。根据定义，于是你可以取得这个被包含对象的地址。

谁给我们带来麻烦

现在的问题是：对标准库中支持 `operator[](0)` 的任何容器来说，例 6-2 中的代码都是合法的；如果去掉包含 “`&t[0]`” 的那行语句，对标准库中的每一个容器来说，它都是合法的，但是，唯独 `std::vector<bool>` 除外。

想知道为什么，请看下面的模板；这个模板可以用于任何类型 T，但是，`bool` 除外。

```
// 例 6-3: 对除 bool 之外的每一个 T 都可行
//
```

```
template<typename T>
void g( vector<T>& v )
{
    T* p = &v.front();
    // ... 使用*p ...
}
```

明白为什么吗？说这段代码对所有的类型都合法，但唯独有一个例外，这让人初次（甚至第二次、第三次）听起来似乎觉得有点古怪。是什么让 `vector<bool>` 如此特殊呢？

原因很简单，因为，不幸的是：并非所有定义在 C++ 标准库中并且看起来像容器的模板都真的是容器。具体来说，C++ 标准库要求 `vector` 必须针对 `bool` 进行特殊化，于是，作为一个模板特殊化，`vector<bool>` 不是一个容器，它不符合标准库容器的条件。是的，`vector<bool>` 的确出现在 C++ 标准的“容器和序列”部分；是的，标准中没有提到它实际上既不是容器也不是序列。但事实是，`vector<bool>` 不是容器，所以，它不总是能够作为一个容器来使用^⑦，对此你不必奇怪。

如果这种状况让你感到有点奇怪，我也可以理解。但这里有一个合理的解释。

`vector<bool>` 揭秘

`vector<bool>` 特殊化往往被 C++ 标准（错误地）作为一个例子，用来演示如何写一个被代理容器。“被代理容器（proxied container）”指的是其对象不能被直接访问和操作的容器。被代理容器不会向你提供被包含对象的指针或引用，而是提供一个代理对象，让你通过这个代理对象间接地访问和操作被包含对象。如果一个集合（译注：“容器”是一种特殊的“集合”）中的对象不能（像被放在内存中那样）被可靠地直接访问，被代理集合（proxied collection）会很有用——例如一个涉及磁盘操作的集合，它在底层自动地调度内存页面，根据需要将自己的数据块在磁盘和内存之间来回交换。所以很多人有一个错误的认识，认为 `vector<bool>` 演示了“如何使一个被代理集合满足标准库所定义的‘容器’条件”。

令人扫兴的是，从满足标准容器和迭代器条件的意义上来说，`vector<bool>` 不是一个“容器”。^⑧ 被代理容器是绝对不会被标准容器和迭代器条件所接受的。例如，

^⑦ 如果是其他什么人写了这么一个 `vector<bool>` 特殊化，我们可以称它是“不符合要求”和“非标准”的；但实际上它是写在 C++ 标准中，所以那么称呼它很难。消除这种不一致性的正确方案是删除 C++ 标准的条款 23.2.5，即，删除 C++ 标准对 `vector<bool>` 特殊化的要求。那样的话，`vector<bool>` 就只不过是 `vector`<主模板的另一个实例化 instantiation>，从而让 `vector<bool>` 名副其实——成为一个包含“普通的旧式 `bool`”的 `vector`。（另外，`vector<bool>` 的功能在许多方面是多余的；`std::bitset` 就是用来完成它那种压缩式表示（packed-representation）的功能的。）

^⑧ 显然，从普通意义上来说它是个容器，因为它是一个可以放入和取出东西的集合。

`container<T>::reference` 必须是一个真实的引用 (`T&`)，绝对不能是一个代理。迭代器也有类似的要求，所以，解引用 (`dereference`) 一个单向 (`forward`)、双向 (`bidirectional`) 或随机访问 (`random-access`) 迭代器必须产生一个真实的引用 (`T&`)，绝对不能是一个代理。有了这些要求，任何基于代理的容器都绝对不会满足 C++ 标准目前所规定的容器条件。

`vector<bool>` 不符合条件的原因在于，它在底层做了些手脚，试图优化存储空间。一个正常的 `bool` 对象的大小至少相当于一个 `char`。`sizeof(bool)` 的定义和具体实现相关，不同的编译器都不一样，但它必须至少为 1。不觉得这样有点浪费吗？一些人确实这么认为，所以他们想让 `vector<bool>` 在空间使用上更高效。因而，`vector<bool>` 没有为它所包含的每一个“`bool`”分配一个完整的 `char` 或 `int`；相反，它在内部表示中将所有的 `bool` 进行压缩存储，让它们每一个各占 1 位 (`bit`)（可以称之为内部 `char` 或 `bool`）。这就给 `vector<bool>` 带来了空间存储上的优势，在一个“正常” `bool` 为 8 位的平台上，它至少节省了 8 倍的存储空间，甚至可能更多。（至此一些人可能已经注意到，这种优化使得 `vector<bool>` 这个名称更加使人误解。因为事实上它的内部根本就不是普通的 `bool`）

这种压缩表示 (`packed representation`) 的一个后果是，`vector<bool>` 显然不能通过它的 `operator[]` 或“被解引用的迭代器”返回一个正常的 `bool&`。^⑨ 相反，它只能返回一个“和引用类似”的代理对象，这个代理对象表面上在很多方面像 `bool&`，但实际上不是 `bool&`。不幸的是，这样一来，对 `vector<bool>` 内部的访问还会变得更慢，因为我们必须得和代理打交道，而不是直接使用指针或引用（更不用说那些额外的位操作了）。所以，`vector<bool>` 不是一种纯粹的优化，而是一种选择，它是用“潜在的速度上的损失”换取“较少的空间占用”。关于这一点，一会儿还要继续分析。

例如，尽管函数 `vector<T>::operator[](0)` 和 `vector<T>::front()` 在正常情况下都简单地返回一个 `T&`（一个引用，指向被包含的 `T`），但 `vector<bool>::operator[](0)` 和 `vector<bool>::front()` 实际返回的是一个类型为 `vector<bool>::reference`、“和引用类似”的代理对象。为了让这个代理对象尽可能地看起来像一个真实的 `bool&`，`reference` 提供了一个隐式转换 `operator bool()`；这样，在可以使用 `bool` 的大多数地方，`reference` 也可以使用——至少可以作为值 (`value`) 使用。此外，它还提供了成员函数 `operator=(bool)`、`operator=(reference&)` 和 `flip()`，这些函数可以用来间接地改变实际存在于容器内部的 `bool` 值，并让我们可以使用“`v[0] = v[1];`”这种很自然的编码风格。（非常有趣的是，注意到有一项功能 `vector<bool>::reference` 肯定无法提供，即，`bool* operator&()`；也就是说，它无法提供一个合适的取址运算符；因为，作为 `vector<bool>` 的代理，`reference` 无法取得 `vector<bool>` 内部单个比特的地址）

^⑨ 因为没有一种标准的方法可以表示“指向比特的指针或引用”。

被代理容器与 STL 假设

结论：或许可以说，`std::vector<bool>`是符合标准的，但它不是个容器。（它也不是个很好的存储比特值的 vector，因为它丧失了一些针对比特的操作功能，而提供这些操作是很合理的，`std::bitset`就提供了这些功能）

此外，最初的 STL 容器条件和 C++ 标准的容器条件都基于一个毋庸置疑的假设（这个假设是众多假设中的一个），即，解引用一个迭代器 (a) 是一个常数时间的操作；

(b) 和其它操作相比，它需要的时间可以忽略不计。对基于磁盘操作的容器或采用压缩表示的容器来说，这两个假设没有一个可以成立。就基于磁盘操作的容器而言，通过代理对象的存取访问需要对磁盘寻道，这会比内存访问慢几个数量级。例如，你一般不会将 `std::find()` 这样的标准算法应用于基于磁盘操作的容器；较之专用算法，其性能会糟糕透顶，这主要是因为：对“内存中的容器”的基本性能假设不适用于“基于磁盘操作的容器”。（基于类似的原因，即使是内存中的容器，例如 `std::map`，也会以成员函数的形式提供它们自己的 `find()`）对于 `vector<bool>` 这种采用压缩表示的容器而言，通过代理对象进行存取访问将需要执行逐位操作（bitwise operation），较之 `int` 这种普通类型的操作，其速度通常会慢很多。此外，如果编译器不能通过优化将代理对象的构造和析构完全消除掉，对代理对象本身的管理也需要增加额外的开销。

在一定程度上，`vector<bool>` 可以作为一个例子，用来演示如何写被代理容器；这样，其他程序员在编写基于磁盘操作的容器时，或是在编写其它什么容器、但容器无法“可靠”、“直接”地访问被包含对象时，他们可以借鉴这个例子。同时，`vector<bool>` 还可以用来说明，被代理容器不符合 C++ 标准目前的容器条件。

被代理集合是个有用的工具，很多场合下使用它都很合适，对于那些会增长得很大的集合来说，尤其如此。每个程序员都应该了解它们。只不过，它们不像很多人所想象的那样适用于 STL，仅此而已。虽然在演示如何写一个被代理容器方面，`vector<bool>` 是一个绝佳的模型，但从 STL 的角度看，它却不是一个“容器”；它应该被称为别的什么东西（一些人建议称之为“bitvector”），这样，人们才不会认为它和一个符合标准的容器有什么联系。

谨防过早优化

如果你阅读过 *Exceptional C++*，对于我反对“过早优化”的老生常谈，你应该已经见多不怪了。^⑩ 那些规则可以总结为：(1) 不要过早优化。(2) 除非知道确实必要，否则不要使用优化。(3) 即便那样，除非已经知道了要优化什么、哪里需要优化，否则也不要使用优化。有些人说得更直截了当：

^⑩ 参见条款 12.

- 使用优化的第一条原则：不要使用它。
- 使用优化的第二条原则：还是不要使用它。

一般来说，对于自己所写的代码在空间和时间性能上的实际瓶颈，程序员（包括你我）的猜测糟糕得一踏糊涂。如果没有性能分析或其它实验数据指导你，你会很轻易地花掉几天的时间去优化一些不需要优化的东西，或者，实际上不影响运行时期空间性能或时间性能的东西。然而更糟糕的是，当你不知道什么东西需要优化的时候，你实际上可能是在借“优化”之名行“恶化”（损害你的程序）之实，因为你可能会为了节省一点成本，却在无意中招致更大的成本。^⑩一旦你进行了性能分析或其它测试，并且确实知道某种优化会对你有帮助，这时候进行优化才是水到渠成之事。

至此，你可能会同意我的看法：“过早优化”的极致莫过于深藏在 C++ 标准中的优化；对于这种优化，我们很难躲避。具体来说，`vector<bool>`实际上是有意用“更慢的执行速度”来换取“更少的存储空间”，并将这一优化选择强加给所有的程序。这种优化选择本质上是在假设：所有使用 `vector<bool>` 的用户都愿意用更慢的速度换取更少的空间，或者，用户在空间上的限制大于性能上的限制，等等。很明显，这种假设是不成立的；在很多很普及的实现中，`bool` 的大小和 8 比特的 `char` 的大小相同，一个包含 1000 个 `bool` 的 `vector` 占用大约 1K 内存。在很多应用程序中，节省这 1K 内存并不重要（是的，节省 1K 空间在某些环境下显然很重要，例如嵌入式系统；还有，一些应用程序操作的 `vector` 可能会包含数百万个 `bool`，在这种情况下，节省数 M 字节的空间当然意义重大。）这里想说明的要点是：正确的优化取决于具体的应用。假设你在写一个程序，你在一个内部循环中操作一个有 1000 个元素的 `vector<bool>`，这种情况下，较之节省少量的空间，由于 CPU 负荷减少（没有代理对象和位操作的开销）所带来的最根本的速度上的提高会更能让你的程序受益——即使空间的节省会提高高速缓存的命中率。

究竟该何去何从？

如果你正在自己的代码中使用 `vector<bool>`，你可能是在将它用于某些算法，而这些算法需要的是真正的容器（因而，这种用法可能可以移植，但也许不能）；同时，你是在使用一种以时间换取空间的优化法（这一点可能不易察觉到）。这两点中很有

^⑩ 关于这一点有一个有趣的例子，请参阅附录 A “（在多线程环境下）并非优化”，那是一个很恰当的佐证。那个附录剖析了一个很普及的优化法，但出人意料的是，那个优化法主要只适合于单线程环境；在（有可能）使用多线程的程序中，它实际上是一种真真切切的恶化行为。

可能会有一点对你的程序不合适。发现第一点的简单方法是：使用这些代码，直至编译出错。但这一天可能永远也不会来临。发现第二点的唯一方法是做试验，针对代码中使用 `vector<bool>` 的方式，我们对其性能进行测试。很多情况下，和 `vector<int>` 这样的东西相比，如果它们之间存在性能上的差异，其差异也很可能微乎其微。

如果你正在为 `vector<bool>` 的“非容器性”所困，或者，如果在你的环境中所测量出来的性能差异并非微不足道，而且这种差异对你的应用程序影响巨大，那么，请不要使用 `vector<bool>`。你可能首先会想到用 `vector<char>` 或 `vector<int>` 来代替它（即，将 `bool` 作为 `char` 或 `int` 来存储），并且，在对容器中的值进行设定时使用类型转换。但这样做很麻烦。更好的解决之道是使用 `deque<bool>`；这是一个更简单的方案，而且它和条款 7 中有关 `deque` 的使用建议相吻合。

总结

(1) `vector<bool>` 不是容器。

(2) `vector<bool>` 试图向你演示如何编写一个“在底层隐蔽地做其它工作”同时“符合标准”的被代理容器。遗憾的是，这不是个好主意，因为，尽管被代理集合是一个重要并且有用的工具，但根据定义，它必然违反 C++ 标准的容器条件，因而绝对不可能成为一个符合标准的容器（见第 1 点）。

推论：继续写你的被代理集合吧，但甭想让它成为一个还满足标准容器或序列要求的容器。首先，这不可能。第二，之所以想让集合符合标准容器的条件，其主要原因是想将其用于标准算法；然而，标准算法一般不适用于被代理容器，因为，较之存在于内存中的普通容器，被代理容器具有不同的性能特征。

(3) `vector<bool>` 的名称有点让人误解，因为其内部元素甚至不是标准的 `bool`。标准的 `bool` 至少具有和 `char` 一样的大小，从而可以被“正常”使用。所以，实际上，`vector<bool>` 甚至没有真正存储 `bool`，它只是徒有其名。

(4) 由于 `vector<bool>` 存在于标准之中，它就将一种特定的优化强加给了所有的用户。这可能不是一个好主意，即使对某个编译器和大多数程序来说，其实际性能开销可以忽略不计；何况，不同的用户有不同的需求。

最后再说一句：明智地使用优化。除非有实验数据证明，在你所在的情况下，优化会给你带来好处，否则，绝对不要屈服于过早优化的诱惑。

条款 7：使用 vector 和 deque

难度：3

`vector` 和 `deque` 有何区别？分别在什么时候使用它们？对于这样的容器，当你不再需要其全部容量的时候，应当如何适当地缩小它们呢？本条款提供了这些问题的答案，以及其他更多话题。你可以将它们看作来自 C++ 标准前沿的最新信息。

(1) 在标准库中，`vector` 和 `deque` 提供了近似的功能。通常情况下你应该选用哪一个？为什么？在哪些场合你会考虑使用另一个？

(2) 下面的代码完成了什么操作？

```
vector<C> c( 10000 );
c.erase( c.begin() + 10, c.end() );
c.reserve( 10 );
```

(3) 通常，`vector` 或 `deque` 会保留额外的内部空间，以备将来增长的需要，从而防止增加新元素时过于频繁地重新分配。有可能完全清空一个 `vector` 或 `deque`（即，不仅要删除所有包含的元素，还要释放所有内部保留的空间）吗？证明为什么可以，或者为什么不可以。

提示：问题 2 和 3 的答案会很微妙。它们都各有一个唾手可得的答案，但不要就此浮于表面；尽可能地详尽分析。

解答

本条款回答了以下问题：

- 可以将一个 `vector` 作为一个 C 风格的数组来使用吗（例如，将 `vector` 用于一个“接收参数为数组”的已知函数）？如何做到？会有什么问题？
- 通常情况下，为什么要优先使用 `vector` 而不是 `deque`？
- 如果一个 `vector` 的容量增长至超过了你的需要，如何去缩小它？具体来说，如何将 `vector` 的内部空间“缩小至合适大小”（使这个 `vector` 的容量恰好足以容纳它当前的元素）或者完全清空（使这个 `vector` 真正为空，即，不包含元素，而且底层预分配的空间几乎没有甚至完全没有）？其它的容器可以使用这一相同的技术吗？

将 `vector` 用作数组：一个启发式的例子

我们强烈鼓励 C++ 程序员使用标准 `vector` 模板，而不要使用 C 风格的数组。使用 `vector` 比使用数组更简单、更安全，因为 `vector` 是一种抽象和封装得更好的容器——例如，根据需要，`vector` 可以增长或缩小；而数组的大小是固定的。

然而，目前还是有大量代码使用了 C 风格的数组。例如，请看下面的代码：

```
// 例 7-1(a)：一个对 C 风格
// 的数组进行操作的函数
//
int FindCustomer(
    const char* szName,      // 要查找的名字
    Customer*   pCustomers, // 指向 Customer 数组
                           // 开始处的指针
    size_t       nLength )  // 数组长度为 nLength
{
    // 执行（可能经过优化的）查找
    // 操作，查找到指定的名字后，
    // 返回其数组下标
}
```

为了使用上面的函数，我们可以像下面这样创建、填充和传递一个数组：

```
// 例 7-1(b)：使用数组
//
Customer c[100];
// --用某种方法添加 c 的内容--
int i = FindCustomer( "Fred Jones", &c[0], 100 );
```

例 7-1(b)完全正确而且运行良好，但作为现代的 C++程序员，我们不是应该尽量使用 vector 而不使用数组吗？的确，如果我们能够做到两全其美——既有 vector 的方便和安全，又可以在接收参数为数组的函数中使用 vector 的元素，这难道不好吗？在一个开发小组从 C 过渡到 C++的过程中，这种需要会经常产生：人们想用“新的更好的”方式来编写系统的新增部分，但还是希望可以很容易地与现有代码库相融合。

那么，我们来使用 vector 吧。我们的第一次尝试可能会像下面这样：

```
// 例 7-1(c)：使用 vector——可行吗？
//
vector<Customer> c;
// --用某种方法添加 c 的内容--
int i = FindCustomer( "Fred Jones", &c[0], c.size() );
```

在例 7-1(c)中，当调用 FindCustomer()时，其基本思想是传入第一个元素的地址以及元素的数量，这和我们在例 7-1(b)中的做法几乎一样。

继续往下阅读之前暂停一会儿，考虑这样的问题：和例 7-1(b)相比，例 7-1(c)的做法可能会有什么好处？你发现例 7-1(c)有什么潜在的问题吗？或者，你认为例 7-1(c)会像你预想的那样运作吗？

美中不足？

例 7-1(c)并非我随意挑选的一个例子。在现实世界中，大量的 C++ 程序员恰恰是这么做的；而且，这样做还基于很好的理由，因为，例 7-1(c)那样的代码可以向我们展示，使用 `vector` 会带来某些重要的好处：

- 我们不必事先知道 `c` 会有多大。在很多 C 程序中，数组是以保守的方式分配的，只有那样，在大多数可以预计的使用场合，它们的容量才够用。遗憾的是，这意味着，在不需要数组全部容量的情况下，这样做大都会造成不必要的空间浪费。（更糟糕的是，这意味着，如果数组实际需要的空间比我们想象的要多，程序还会出问题）有了 C++ 的 `vector`，我们就能让容器“根据需要”增长，而不是“根据猜测”。而且，如果我们刚好事先知道将要把多少对象放进 `vector`，我们可以简单地表达出来（如果想为 100 个元素预留空间，我们可以先调用 `c.reserve(100)`）；所以，和数组相比，`vector` 没有性能上的劣势，因为在向 `vector` 插入新元素时，我们有办法避免重复不断的内存分配。
- 我们不必为了向 `FindCustomer()` 传递最后一个 (`nLength`) 参数而专门去跟踪数组的实际长度。是的，你可以使用 C 风格的数组，并用什么方法更容易地记住或计算出实际长度，^⑫ 但那些方法都不会像只用写一个 `c.size()` 那样简单安全。

一言以蔽之，就我所知，对于曾经出现过的所有 `std::vector` 实现，例 7-1(c)都可行。

但在 2001 年之前有一个美中不足：实际上，1998 年制定的 C++ 标准[C++98]没有保证例 7-1(c)会在你可能购买的每一个 C++ 平台上可行。这是因为，例 7-1(c)假设 `vector` 的内部元素是连续存储的，而且存储格式和数组相同；但最初的 C++ 标准并没有要求供货商以那样的方式实现 `vector`。例如，如果某个 `vector` 的实现标新立异地用其它某种方式存储内部元素（比如，虽然是连续存储但次序相反；或者，虽然是顺序存储但在元素之间带有额外的字节，用以存储内部管理信息），那么，例 7-1(c)事实上会悲惨地以失败告终。2001 年，在 C++ 标准第一号技术勘误表（Technical Corrigendum）中，这一问题已经得以定案；现在，C++ 标准要求，`vector` 的内部元素必须像数组那样连续存储。

即使你的编译器尚未包含对技术勘误表的支持，但实际上，例 7-1(c)在目前是可行的。再次说明，我没有看到任何一个商业性的 `vector` 实现不是连续存储元素的。

结论：尽量使用 `vector`（或 `deque`，参见下文），不要使用数组。

^⑫ 最典型的方法是为数组大小提供一个 `const` 值或一个`#define` 名称；或者，提供一个宏，其展开式为 `sizeof(c)/sizeof(c[0])`，用以计算数组的大小。

在大多数场合下，尽量使用 vector

1. 在标准库中，vector 和 deque 提供了近似的功能。通常情况下你应该选用哪一个？为什么？在哪些场合你会考虑使用另一个？

关于应该优先使用何种容器，C++标准[C++98]第 23.1.1 节提供了一些建议。它是这么说的：

vector 是那种应该在默认情况下使用的序列……当大多数插入和删除操作发生在序列的头部或尾部时，应该选用 deque 这种数据结构。

vector 和 deque 提供了几乎相同的接口，因而往往可以互换。deque 还提供了 push_front() 和 pop_front()，这是 vector 所没有的。（是的，vector 提供了 capacity() 和 reserve()，deque 则没有，但这对 deque 来说没有损失——过一会儿我会证明，那两个函数实际上反映了 vector 的某种弱点）

vector 和 deque 在结构上的主要区别在于，这两种容器组织内存的方式不一样。在底层，deque 是按“页”（page）或“块”（chunk）来分配存储器的，每页包含固定数目的元素；这就是为什么 deque 常常被比喻为（并发音为）a “deck” of cards（一副牌），^⑩ 尽管它的名字最初源于“double-ended queue”（双端队列）——因为它在序列的两端插入元素都很有效率。相反，vector 分配一块连续的内存，只是在序列的尾端插入元素时才有效率。

deque 的分页组织方式提供了几个好处：

- 即使在容器的前端，deque 也提供了常数时间的 insert() 和 erase() 操作，而 vector 则不行；所以 C++ 标准才注明，需要在序列的两端进行插入或删除操作时，应该使用 deque。
- deque 使用内存的方式对操作系统更友好。例如，一个 10M 字节的 vector 使用的是一整块 10M 字节的内存。在一些操作系统上，那一整块内存会比一个 10M 字节的 deque 缺乏效率，因为 deque 可以使用一串更小的内存块。在其它某些操作系统上，一段连续的地址空间可以在底层通过更小的内存块组成；对于这样的系统，deque 非连续性的优点也不会受影响。
- deque 更容易使用一些，而且在体积增长方面，它天生比 vector 更有效率。vector 提供了两个函数 capacity() 和 reserve()，这是仅有的两个由 vector 提供但 deque 没有提供的操作。这是因为 deque 不需要它们！对 vector 来说，如果在执行大量的 push_back()

^⑩ 就我所知，将 deque 比喻为“deck of cards”最早是由 Donald Knuth 在[Knuth97]中提出的。

操作之前不调用 `reserve()`，那么，当它每次发现当前内存不够的时候，都会对同一块缓冲区重新分配，以获得更大的内存；如果使用 `reserve()`，就可以消除这种重新分配的情况。`deque` 不存在这样的问题，在执行大量的 `push_back()` 操作之前调用一个 `deque::reserve()` 不会消除任何分配（或任何其它）操作，因为它没有多余的分配操作。`deque` 必须分配相同数量的额外内存页，无论是立即分配，还是在元素实际添加时分配。

有趣的是，请注意到，在 C++ 标准本身中，`stack` 适配器优先使用的是 `deque`。尽管 `stack` 只能在一个方向上增长，因而永远不需要在中间或另一端进行插入，但它的缺省实现使用的是 `deque`：

```
namespace std
{
    template<typename T, typename Container = deque<T>>
    class stack {
        // ...
    };
}
```

尽管如此，为了可读性和简单化，在默认情况下，请尽量在程序中使用 `vector`，除非你需要在容器的头部执行有效的增加或删除操作，并且不需要底层对象连续存储。请注意，`vector` 之于 C++，正如数组之于 C——它应当是你的程序中使用的默认容器类型，它可以相当出色地帮助你完成大多数工作，除非你知道自己需要的是与之不同的东西。

不可思议地缩小的 `vector`

正如前面所说的那样，`vector` 管理它自己的内存；而且，针对将来增长的需要，我们可以告诉 `vector` 应该在底层准备多大的容量。如果有一个 `vector<T> c`，我们打算让它的元素增长到 100 个，我们可以先调用 `c.reserve(100)`，这样，最多就只会带来一次再分配。这就让增长操作的效率达到最理想的程度。

但如果我们要做相反的事呢？如果我们正在使用一个特别大的 `vector`，然后我们想删除不再需要的元素，使 `vector` 缩小到合适的大小，那该怎么办？——也就是说，我们想清除目前不再需要的空间，该怎么办？你可能认为下面的方法可以完成这一任务：

2. 下面的代码完成了什么操作？

```
// 例 7-2(a): 缩小 vector 的一个幼稚的尝试
//
vector<C> c( 10000 );
```

现在，`c.size() == 10000`，并且`c.capacity() >= 10000`。

接着，我们删除`c`中除前 10 个元素之外的所有元素：

```
c.erase( c.begin() + 10, c.end() );
```

现在，`c.size() == 10`，而且我们知道，我们不再需要那些额外空间，但下面这行代码没有将`c`的内部缓冲区缩小至合适大小：

```
c.reserve( 10 );
```

例 7-2(a)的最后一行代码没有完成你所期望的功能，原因在于，调用`reserve()`永远不会缩小`vector`的容量。调用`reserve()`只能增加容量；或者，如果容量已经足够，它就什么事都不做。

幸运的是，有一个正确的方法可以让你达到想要的效果：

```
// 例 7-2(b): 将 vector 缩小至合适大小的正确方法
//
vector<Customer> c(10000);
// 现在, c.capacity() >= 10000...

// 删除前 10 个元素之外的所有元素
c.erase( c.begin() + 10, c.end() );

// 下面一行代码真的将 c 的内部缓冲区
// 缩小至合适（或近似）大小
vector<Customer>( c ).swap( c );

// 现在, c.capacity() == c.size(),
// 或者稍稍大于 c.size()
```

那条“缩小至合适大小”的语句是如何工作的？你看明白了吗？确实有点微妙：

(1) 首先，我们创建一个临时（未命名）的`vector<Customer>`并对其初始化，使之包含与`c`相同的内容。临时`vector`和`c`之间最显著的不同是，当`c`还在其过大的内部缓冲区里背负着大量额外空间的时候，临时`vector`所具有的容量却刚好足以容纳`c`所有元素的拷贝。（一些`vector`实现会通过上一个舍入（round up）对容量做微小的调整，使其等于下一个较大的内部“存储块大小”，这样，最终的实际容量会比原始容量稍大。）

(2) 然后，我们调用`swap()`，将`c`的内部缓冲区与临时`vector`交换。现在，临时`vector`拥有的是那块过大的缓冲区，它携带着我们想要删除的那块额外空间；而`c`拥有的是容量刚好足够、具有“正确大小”的缓冲区。

(3) 最后，临时`vector`离开生存空间，带走了那块过大的旧缓冲区。临时`vector`被摧毁时，旧缓冲区被删除。现在，剩下的只有`c`本身，而`c`现在拥有的是具有“正确大小”的容量。

注意，这一过程没有导致不必要的效率损失。哪怕 `vector` 会提供一个专用的 `shrink_to_fit()` 成员函数，它也得完成几乎所有以上描述的工作。

3. 通常，`vector` 或 `deque` 会保留额外的内部空间，以备将来增长的需要，从而防止增加新元素时过于频繁的重新分配。有可能完全清空一个 `vector` 或 `deque`（即，不仅要删除所有包含的元素，还要释放所有内部保留的空间）吗？证明为什么可以，或者为什么不可以。

如果想完全消除一个 `vector`，使它不包含任何元素并完全没有额外容量，代码几乎和前面相同。我们只需要将临时 `vector` 初始化为空，而不要使其成为 `c` 的拷贝：

```
// 例 7-3: 清空 vector 的正确方法
//
vector<Customer> c(10000);
// 现在, c.capacity() >= 10000...

// 下面一行代码使得
// c 真的为空
vector<Customer>().swap( c );

// ...现在, c.capacity() == 0,
// 除非 vector 实现强制让空 vector
// 包含某一最小容量
```

再次注意，你所使用的 `vector` 实现可能会让空 `vector` 也拥有少量的容量，但现在可以保证，`c` 的容量是你的编译器所能允许的最小值，它和一个空 `vector` 的容量是一样的。

这些技术对 `deque` 也一样奏效，但你无需对 `list`、`set`、`map`、`multiset` 或 `multimap` 做类似的事，因为它们基于“严格遵循所需”的原则分配空间，因而永远不会存在多余的容量。

总结

用 `vector` 替换 C 风格的数组在使用中是安全的，而且你应该总是尽量使用 `vector` 或 `deque`，而不使用数组。将 `vector` 作为你的默认容器类型使用，除非你知道自己需要的是不同的东西，并且不需要所有被包含元素在内存中连续存储。如果想缩小一个已有的 `vector` 或 `deque`，请运用“和一个临时容器交换”的手法。最后，正如条款 6

所说的那样，除非你真的需要空间优化，否则请总是使用 `deque<bool>`，而不要使用 `vector<bool>`。

条款 8：使用 set 和 map

难度：5

在条款 7 中，我们讨论了 `vector` 和 `deque`。这一次，我们将注意力更多地放在关联式容器 `set`、`multiset`、`map` 和 `multimap` 身上。^④

1. a) 下面的代码有什么错误？如何改正？

```
map<int, string>::iterator i = m.find( 13 );
if( i != m.end() )
{
    const_cast<int&>( i->first ) = 9999999;
}
```

b) 上面的代码改为下面这样，问题解决了多少？

```
map<int, string>::iterator i = m.find( 13 );
if( i != m.end() )
{
    string s = i->second;
    m.erase( i );
    m.insert( make_pair( 9999999, s ) );
}
```

可以通过 `set::iterator` 修改 `set` 的内容吗？为什么可以？或者，为什么不可以？

解答

关联式容器：回顾

首先，快速地复习一下。下面是一些要点，涉及的内容有：对关联式容器（associative container）进行遍历是什么意思？什么是关联式容器的迭代器？它能做些什么？

^④ 注意，`set` 和 `multiset` 并不真正具有普通意义上的“关联”含义，因为它没有像字典那样，将一个查找键（key）和另一个值（value）联系起来。那是 `map` 和 `multimap` 的功能。但在 C++ 标准中，这四个容器全都出现在“关联式容器”的标题下，所以我也这样做，以保持一致。

图 1 汇总了四种标准关联式容器。每种容器都在内部以一定的方式保存它的元素，使得我们可以按照键（key）的非降序方式快速遍历元素，或通过键快速查找元素，其中，对键的比较总是依据所提供的 Compare 类型来进行的（Compare 缺省为 `less<Key>`，它针对 Key 对象执行 `operator<()` 操作）。只要有一个新键插入容器，容器就会找到一个合适的位置存放这个新键，从而使内部数据结构保持正确的次序。

迭代器很容易实现，因为它们应该以键的非降序方式遍历元素，而在容器的内部，元素的存储方式已经自然支持这种次序。太基本的知识了，对吗？

```
set<Key, Compare>
multiset<Key, Compare>
map<Key, Value, Compare>
multimap<Key, Value, Compare>
```

图 1：标准关联式容器（省略了空间配置器参数）

那么，让我们看看“重要”议题：

重要条件

关联式容器依赖一个基本条件，缺少了它，容器根本就不能可靠工作，这个基本条件是：一旦一个键被插入容器，那么，无论那个键被什么方式修改，它在容器中的相对位置不能改变。如果这种事真的发生，容器会毫不知情，它对元素次序的假设将被推翻，查找合法元素的操作会失败，迭代器就不一定能按照键的次序遍历容器的元素，总之，麻烦事就会发生。

我将通过一个例子来说明这个问题，然后讨论如何处理这种情况。

一个启发式的例子

请看一个名为 m 的 `map<int,string>`，它包含的元素如图 2 所示；m 中的每个节点以 `pair<const int, string>` 表示。我将 m 的内部结构表示为一个二叉树，因为目前所有的标准库的实现也确实是这么做的。

插入键时，树的结构继续保持平衡，所以，一个正常的中缀次序的遍历（inorder traversal）会以普通的 `less<int>` 次序对键进行访问。至此，一切都还不错。

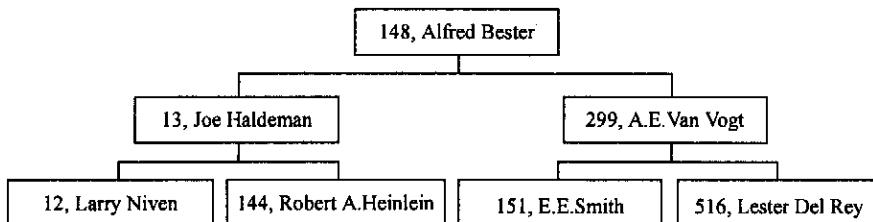


图 2：一个 map<int, string>的内部结构示例

但现在假设，通过一个迭代器，我们可以任意修改第 2 个元素的键，代码像下面这样：

1. a) 下面的代码有什么错误？如何改正？

```

// 例 8-1：修改 map<int, string> m
// 中一个键的错误做法
//
map<int, string>::iterator i = m.find( 13 );
if( i != m.end() )
{
    const_cast<int&>( i->first ) = 9999999; // 噢!
}
  
```

注意，要想让代码通过编译，我们必须转换掉 `const`。关于这一点后面还会讨论。目前的问题是，这段代码侵犯了 `map` 的内部表示，它改变 `map` 内部表示的方式对 `map` 来说无法预计，也无法应对。

例 8-1 破坏了 `map` 的内部结构（见图 3）。举例来说，如今，在用迭代器进行遍历时，将不会按照键序返回 `map` 的元素，而它本应该如此。例如，查找键 144 可能会失败，即使这个键在 `map` 中存在。一句话，这个容器不再处于稳定或可用的状态。注意，要想防止这样的非法使用，靠 `map` 自身的能力是不可能的，因为当这种修改行为发生时，它甚至无从知晓。在例 8-1 中，修改操作是通过一个指向容器的引用进行的，并没有调用任何一个 `map` 的成员函数。

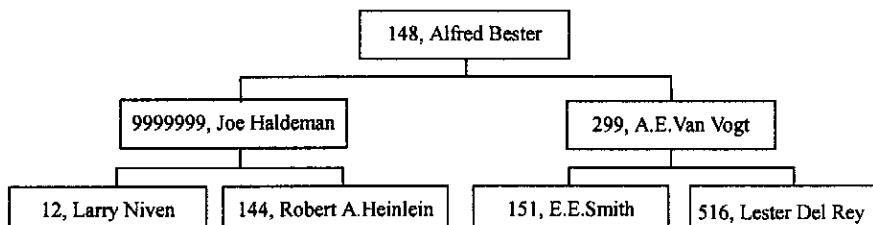


图 3：例 8-1 执行后，图 2 的变化结果。出现问题！

在例 8-1 的代码中，如果在对键进行修改时，所采用的方式不会导致键的相对次序发生改变，这个 map 实现就不会出问题。键一旦存在于容器中，就有了相关次序，只有试图修改这一次序的代码才会产生问题。

解决这个问题的最好方法是什么？理想情况下，我们可以通过合适的编程规范来防止这种情况。那么，这样一种编程规范应该有怎样的规定呢？

选择 1：规定“**const 就是 const!**”（不充分）

在例 8-1 中，为了对键进行修改，我们需要转换掉 const。这是因为，在防止代码修改“键的相关次序”方面，标准 C++ 做得很积极。实际上，标准 C++ 制定了一条（似乎）更严格的规定——即，对于 map 和 multimap，键根本不应该被修改。`map<Key,Value>::iterator` 指向的是 `pair<const Key, Value>`，显然，它只让你修改所指向的 map 元素的“值”部分，而不是“键”部分。很显然，这完全防止了对键的修改，因而更不存在某种方式，可以改变键在 map 对象的内部排序中的位置。

因此，一种做法是，印刷一张很大的海报，上面写上“**const 就是 const!**”，并且召集著名演讲人或鼓动大众媒体进行宣传，从而让人们对这一应该受到严厉谴责的问题提高认识。对于例 8-1 那种实际上不正确的代码，这种做法的确可以起到预防作用，不是吗？

这是个好主意，但不幸的是，就算告诉人们去尊重 const，这还不够。例如，如果 Key 类型有一个 mutable 成员，这个成员影响 Compare 对 Key 对象作比较的方式，那么，在一个 const 键上调用 const 成员函数还是可以改变键的相关次序的。

选择 2：规定“总是以‘先删除再重新插入’的方式对键进行修改”（好一些，但还是不充分）

有一个更好的方案，但还是不充分；这个方案基于这条规则：要对一个键进行修改，先删除它，然后再重新插入。例如：

b) 上面的代码改为下面这样，问题解决了多少？

```
// 例 8-2：修改 map<int, string> m
// 中一个键的更好的方式
//
map<int, string>::iterator i = m.find( 13 );
if( i != m.end() )
{
    string s = i->second;
    m.erase( i );
    m.insert( make_pair( 9999999, s ) ); // 没问题
}
```

这个方案好一些，因为它可以避免对键的任何修改——即使键具有 `mutable` 成员，而且这些成员对于排序很重要。对于我们这个特定的例子，它甚至是可行的。那么，这必然可以成为一个方案了，对吗？

很遗憾，它还是不足以应用到一般场合，因为在容器内部，键还是可以被修改。“什么？”有人会问：“如果我们遵守规定，永远不直接修改键对象，那么，在容器的内部，键怎么可能被修改呢？”这里是两个反例：

(1) 假设 `Key` 类型具有某种外部可见的状态，这个状态值可以被其它代码得到——例如一个指针，指向共享缓冲区，系统的其它部分无需操作 `Key` 对象就可以修改这一缓冲区。我们再假设，那个外部可见的状态值参与了 `Compare` 执行的比较操作。那么，即使对 `Key` 对象一无所知，即使对使用这个关联式容器的代码一无所知，如果对外部可见的状态值进行修改，就还是可以改变键的相关次序。所以，在这种情况下，即使拥有那个关联式容器的代码试图遵循“先删除再重新插入”的规则，但在任何时候，键序的改变还是可以在别的地方发生——因而，也就无法实现这种“先删除再重新插入”的操作。

(2) 假设 `Key` 的类型为 `string`；`Compare` 类型将这个 `Key` 解释为“文件名”，在做比较时，比较的对象是文件的“内容”。在这种情况下，显然，即使键永远没有被修改，但如果文件（译注：指文件内容）被另一个进程修改，或者（如果文件被共享在网络上的话）被世界另一端的另一台计算机上的某个用户修改，键的相关次序还是可以被改变。

关于 Set

再来看看 `set`。因为还没有为 `map` 找到一个好的答案，我们也想看看 `set` 的情况怎样。

2. 可以通过 `set::iterator` 修改 `set` 的内容吗？为什么可以？或者，为什么不可以？

对于 `set`，我们可以把它看作一个 `map<Key, Value>`，但 `Value` 的类型为 `void`。你可能会想，`set` 处理键类型的方式会和 `map` 的做法大同小异，`set<Key>::iterator` 和 `set<Key>::const_iterator` 也会是一回事——即，二者指向的都是 `const Key`，这样一来，就可以防止程序员对键进行修改。

可惜，在这一点上，C++ 标准没有像对 `map` 那样，也对 `set` 规定得很清楚；但是，想对 `set` 内部的某个对象进行修改是很合理的——毕竟，内部对象可能包含其它信息，而这些信息可能不影响比较操作。关于 `set::iterator` 是否应该指向非 `const` 键，曾经存在意见截然相反的两个阵营；所以，在一些标准库实现中会这样做，在另一些标准库实现中不会这样做，这一点都不奇怪。换句话说，在实际工作中，想通过不带 `const_cast` 的 `set::iterator` 去修改 `set` 中的元素，这种做法不具可移植性。在本书交付出版之际，标准委员会内部的

意向是，在标准中增加一些措词，正式要求 `set::iterator` 和 `set::const_iterator` 必须是常量迭代器（译注：即 `constant iterator`，它是一种“无法用来改变其所指之值”的迭代器，但迭代器本身可能为 `const`，也可能不是。参阅 Austern 的 *Generic Programming and the STL* 第 2.3 节），因而，如果想修改 `set` 中的对象，你就必须使用 `const_cast` 这一转换工具。

然而，有很好的理由使用这种转换工具。请记住，使用 `set` 和 `map` 时，其目的通常有点不同。例如，假设你想实现一个查找功能，即，给出一个顾客的名字，可以查找出顾客的地址或电话号码。于是，你会通过一个 `map<string, CustData>` 来实现，其中，“键”为顾客的名字，“值”为一个 `CustData` 结构，包含地址和电话号码信息。

但如果你已经有一个非常合适的 `Customer` 类，它包含顾客的名字和信息，这种情况下该怎么做？如果再去写一个 `CustData` 结构，使之包含除名字字符串之外的所有相同信息，这会带来不必要的复杂性不说，似乎还是一种开发时间上的浪费。另一方面，如果去实现一个 `map<string, Customer>`，其中键为顾客的名字，那么，由于名字已经存在于 `Customer` 对象中，这个键实际上是一个冗余拷贝，从而带来运行时期的资源浪费。我们确实不应该将名字存储两次。因而，你应该很自然地想到去实现一个 `set<Customer>`，这样做无需新的数据结构，也不会带来冗余的 `string` 开销。对 `map<string, Customer>` 来说，一旦 `Customer` 对象存在于容器之中，这个对象就可以被修改——`set<Customer>` 也应该这样做；但在现实中（可能很快就有正式规定），要想这样做，你必须使用 `const_cast`。目前的一些 `set` 实现允许你不用转换就可以修改它，一些则不允许。如果你使用的 `set` 实现允许修改，无论它要求必须使用 `const_cast` 还是无需使用 `const_cast`，请记住：在修改 `Customer` 对象时，你还是不能改变它在容器中的相关次序。

注意：为了制定更明确的规则，我们在前面做了两个尝试，但它们都不能覆盖所有的情况；还要注意到，即使是 `set` 和 `map` 本身，它们处理事情的方式也有不同。或许，我们所能做的只能是去寻找一条一般性的规则。

重要条件（再论）

为确保正确使用关联式容器，我们应该遵循怎样的规则呢？如果能制定一条更明确的规则，那当然好，但 `const` 无法独立承担使命（不管怎么说，目前的现实是，对于“键的 `const` 性”这一问题，`set` 和 `map` 在策略上存在分歧），简单的编程规范也无法独立承担使命。所以，我们真的无法做得更明确，只能陈述条件本身，并加上一句：“你必须清楚自己在做什么。”

关联式容器的重要使用规则

一个键一旦被插入到关联式容器中，那个键在容器中的相对位置绝对不能改变。

一定要知道哪些直接或间接的行为会改变一个键的相关次序，并且避免这样的行为。这样，在使用标准关联式容器时，就可以避免不必要的问题发生。

条款 9：等同的代码吗？

难度：5

代码中微小的差别真的会带来影响吗？特别是，在“后面填入一个函数参数”这样的简单操作中也是如此吗？本条款就这类问题中一种很有趣的影响进行了探讨——这种影响在 STL 风格的代码中很重要。

(1) 描述下面的代码做了些什么：

```
// 例 9-1
//
f( a++ );
```

请回答得尽可能地完整，涵盖所有的可能性。

(2) 下面两段代码有区别吗？如果有，区别在哪儿？

```
// 例 9-2(a)
//
f( a++ );
// 例 9-2(b)
//
f( a );
a++;
```

(3) 将问题 2 简化，假设 f() 是一个函数，它通过传值的方式获得参数；a 是一个类 (class) 对象，它提供了一个具有正常语义的 operator++(int)。现在再请回答，例 9-2(a) 和例 9-2(b) 有区别吗？如果有，区别在哪儿？

解答

1. 描述下面的代码做了些什么：

```
// 例 9-1
//
f( a++ );
```

请回答得尽可能地完整，涵盖所有的可能性。

包罗万象的列表会让人畏惧，这里只是列出了主要的可能性：

首先，`f` 可以是下面中的任何一个：

1) 宏

这种情况下，这条语句可以表示几乎任何东西，`a++` 可以被求值多次，或者根本就不做什么。例如：

<code>#define f(x) x</code>	<code>// 1 次</code>
<code>#define f(x) (x, x, x, x, x, x, x, x)</code>	<code>// 9 次</code>
<code>#define f(x)</code>	<code>// 什么也不做</code>

设计准则

避免使用宏。宏往往使得代码更难以理解，从而更难以维护。

2) 函数

这种情况下，首先，`a++` 会被求值，然后，结果被传递给函数作为参数。通常，后置递增运算会以临时对象的形式返回 `a` 的旧值，所以，`f()` 获取参数的方式要么是通过传值，要么是通过传递 `const` 引用，但不会是传递非 `const` 引用，因为非 `const` 引用不能被绑定于临时对象。

3) 对象

这种情况下，`f` 将是一个函数对象，即，一个定义了 `operator()()` 的对象。同样，如果后置递增运算返回 `a` 的旧值（后置递增运算总会这样做），那么，`f` 的 `operator()()` 可以通过传值或者传递 `const` 引用来获取参数。

4) 类型名称

这种情况下，这个语句首先对 `a++` 求值，然后用这个表达式的结果来初始化一个类型为 `f` 的临时对象。

再往下看，`a` 可以是：

1) 宏

同样，在这种情况下，`a` 可以表示任何东西。

2) 对象（有可能是内建类型）

这种情况下，它的类型必须定义了合适的后置递增运算符 `operator++(int)`.

通常，后置递增应该利用前置递增来实现，并且返回 `a` 的旧值：

```
// 后置递增的规范形式：
T T::operator++(int)
{
    T old( *this );      // 记住旧值
    ++*this;              // 总是用前置递增
    // 来实现后置递增

    return old;           // 返回旧值
}
```

当然，在重载一个运算符时，你的确可以改变它的正常语义，让它做些“与众不同的事”。例如，假设 `a` 的类型为 `A`，那么，对大多数类型的 `f` 来说，下面的做法很可能使得例 9-1 的代码无效：

```
void A::operator++(int) // 不返回任何值
```

不要这样做。相反，请遵循下面这条有效的建议：

设计准则

始终要为被重载的运算符保留正常语义。“像 `int` 那样做！”；即，遵循内建类型的语义。[\[Meyers96, 条款 32\]](#)

3) 值，例如地址

举例来说，`a` 可以是一个指针。

副作用带来的某些后果

在余下问题中，我们做一些简化性的假设，即，`f()` 不是宏，`a` 是一个对象且具有正常的后置递增语义。

2. 下面两段代码有区别吗？如果有，区别在哪儿？

```
// 例 9-2(a)
//
f( a++ );
```

例 9-2(a)的执行步骤如下：

- (1) a++：递增 a 并返回 a 的旧值。
- (2) f()：将 a 的旧值传递给 f()，然后执行 f()。

例 9-2(a)保证，在 f()执行前，后置递增操作已经完成，因而 a 获得新值。正如前面所说，f()还是可以是一个函数、一个函数对象，或一个导致构造函数调用的类型名称。

一些编程标准规定，++这样的操作应该总是出现在单独一行代码中，其理由在于，考虑到程序执行顺序点（sequence points，更多介绍详见条款 20 和 21）方面的原因，在同一个语句中执行多个++这样的操作很危险。所以，这些编程标准推荐例 9-2(b)这样的做法：

```
// 例 9-2 (b)
//
f( a );
a++;
```

这个例子的执行步骤如下：

1. f()：将 a 的旧值传递给 f()，然后执行 f()。
2. a++：递增 a 并返回 a 的旧值，旧值随后被忽略。

两种情况下，f()得到的都是 a 的旧值。“那么，有什么很大的区别呢？”你会问。噢，是这样：例 9-2(b)不会总和例 9-2(a)一样具有相同的效果，因为例 9-2(b)保证，在 f()执行之后才会执行后置递增，从而使得 a 得到新值。

这有两个主要后果。首先，在 f()产生异常的情况下，例 9-2(a)会保证 a++和它所有的副作用都成功执行结束；而例 9-2(b)则保证 a++没有执行，它的副作用一个也没有发生。

其次，即使在没有异常产生的前提下，如果 f()和 a.operator++(int)有可见的副作用，那么，它们的执行顺序会很重要。更明确地说，如果 f()有一个副作用，这个副作用可以影响 a 本身的状态，考虑一下那将会发生什么事。这个假设并不牵强，也不是不可能：即使 f()不直接改变 a，或者不能够直接改变 a，这也可能会发生，正如下面的例子所演示的那样：

剪刀，交通和迭代器

3. 将问题 2 简化，假设 f()是一个函数，它通过传值的方式获得参数：a 是一个类（class）对象，它提供了一个具有正常语义的 operator++(int)。现在再请回答，例 9-2(a)和例 9-2(b)有区别吗？如果有，区别在哪儿？

区别在于，对于完全正规的 C++ 代码来说，当例 9-2(b) 不合法的时候，例 9-2(a) 可以是合法的。这是因为，在例 9-2(a) 中，有一段时期，对象会同时具有 a 的旧值和新值，而例 9-2(b) 则没有这样的重叠。

如果我们将 f()换成 list::erase()，将 a 换成 list::iterator，考虑一下将会发生什么。这样的话，第一种形式是合法的：

```
// 例 9-3(a)
//
// l 是一个 list<int>
// i 是一个合法的非末端迭代器，指向 l 内部
//
l.erase( i++ ); // 没问题，递增一个有效的迭代器
```

但第二种形式不合法：

```
// 例 9-3(b)
//
// l 是一个 list<int>
// i 是一个合法的非末端迭代器，指向 l 内部
l.erase( i );
i++;           // 错误，i 不是一个有效的迭代器
```

例 9-3(b) 不正确的原因在于，调用 l.erase(i) 使得 i 失效，因而，你随后不再能够对 i 调用 operator++()。

警告：某些程序员的确会写出像例 9-3(b) 那样的代码，这可能是因为，编程准则有一条一般性的原则，不鼓励在函数调用语句中执行++这样的操作。写出例 9-3(b) 那种代码的程序员甚至经常那样做（而且没有意识到危险）但从没出过问题，这只不过是因为，在他们的编译器和程序库的当前版本上，它碰巧可行。但我们得告诫这些程序员：例 9-3(b) 这样的代码不具有可移植性；它没有被标准认可；当你想移植到另一个编译器平台时，或者，甚至只是升级目前工作平台的时候，它很可能会反咬你一口。当它真的咬你的时候，它会咬得很重，因为，“使用无效迭代器”的错误极难发现——除非你乐于在调试期间和一个合格并且出色的程序库实现打交道；但如果真是那样的话，它早就该对这种错误向你提出警告。

一些母亲（同时也是软件工程师）提出了下面三条忠告；我们应当好自为之，努力做到遵循这些忠告：

- (1) 不要摆弄剪刀。
- (2) 不要在交通道路玩耍。
- (3) 不要使用无效的迭代器。

说到这儿, 请注意, 除了像上面这样的例子之外, 在条款 20 和 21 中我们将看到, 一般情况下, 避免在函数调用中写出`++`这样的操作还是个好主意。

条款 10: 模板特殊化与重载

难度: 6

如何对模板进行特殊化和重载? 在这样做的时候, 你如何确定哪个模板会被调用? 动手分析这里的 12 个例子。

(1) 什么是模板特殊化? 给出一个例子。

(2) 什么是部分特殊化? 给出一个例子。

(3) 请看下面的说明:

```
template<typename T1, typename T2>
void g( T1, T2 );                                // 1
template<typename T> void g( T );                  // 2
template<typename T> void g( T, T );                // 3
template<typename T> void g( T* );                 // 4
template<typename T> void g( T*, T );               // 5
template<typename T> void g( T, T* );               // 6
template<typename T> void g( int, T* );             // 7
template<> void g<int>( int );                   // 8
void g( int, double );                            // 9
void g( int );                                    // 10
```

以下每条语句分别调用上面的哪个函数? 正确识别出合适的模板参数类型。

```
int          i;
double       d;
float        f;
complex<double> c;

g( i );           // a
g<int>( i );    // b
g( i, i );       // c
g( c );          // d
g( i, f );       // e
g( i, d );       // f
g( c, &c );       // g
g( i, &d );       // h
g( &d, d );       // i
```

```
g( &d );    // j
g( d, &i );    // k
g( &i, &i );    // l
```

解答

在 C++ 中，模板提供了最强大的通用性（genericity）。有了它，你写出的泛型代码可以和多种互不相关的对象协同工作——例如，内含各种字符的字符串、可以容纳任意类型的对象的容器、可以对任意类型的序列进行操作的算法。

1. 什么是模板特殊化？给出一个例子。

模板特殊化（template specialization）使得模板可以针对特殊情况进行处理。有时，针对某种特定序列（例如，提供了随机访问迭代器），一个泛型算法可以工作得更有效率；所以，如果能针对那种情况进行特殊处理，同时在其它所有情况下仍使用较慢但更通用的方法，那将会很有意义。性能是特殊化的常见理由，但不是唯一理由。例如，某些对象可能不符合通用模板所要求的正常接口，那么，你可以特殊化这个模板，使之可以和那些对象协同工作。

处理这些特殊情况时，你可以使用两种形式的模板特殊化：显式特殊化和部分特殊化。

显式特殊化

显式特殊化（explicit specialization）可以让你针对模板参数的特定组合写出特定的实现。例如，请看下面这个函数模板：

```
template<typename T> void sort( Array<T>& v ) { /*...*/ };
```

如果有某种速度更快的（或其它特殊的）方法来处理 `char*` 类型的 `Array`，我们可以针对这种情况显式地将 `sort()` 特殊化：

```
template<> void sort<char*>( Array<char*>& );
```

于是，编译器会选择出最合适的模板。例如：

```
Array<int> ai;
Array<char*> apc;

sort( ai );      // 调用 sort<int>
sort( apc );     // 调用经过特殊化的 sort<char*>
```

部分特殊化

2. 什么是部分特殊化？给出一个例子。

只有类模板（class template）才能定义部分特殊化（partial specialization），部分特殊化不必处理（未被特殊化的）主类模板（primary class template）的所有参数。

这是选自 C++ 标准[C++98]14.5.4 节[temp.class.spec]的例子。第一个模板是主类模板：

```
template<typename T1, typename T2, int I>
class A { }; // #1
```

我们可以将其特殊化，用于 T2 是一个 T1* 的情况：

```
template<typename T, int I>
class A<T, T*, I> { }; // #2
```

或者，用于 T1 是任何指针的情况：

```
template<typename T1, typename T2, int I>
class A<T1*, T2, I> { }; // #3
```

或者，T1 为 int、T2 为任何指针、I 为 5 的情况：

```
template<typename T>
class A<int, T*, 5> { }; // #4
```

或者，T2 是任何指针的情况：

```
template<typename T1, typename T2, int I>
class A<T1, T2*, I> { }; // #5
```

声明 2 至 5 声明了主模板的部分特殊化。于是，编译器会选择出合适的模板。以下摘自[C++98]第 14.5.4.1 节：

```
A<int, int, 1> a1; // 使用 #1

A<int, int*, 1> a2; // 使用 #2, T 为 int,
// I 为 1

A<int, char*, 5> a3; // 使用 #4, T 为 char

A<int, char*, 1> a4; // 使用 #5, T1 为 int,
// T2 为 char,
// I 为 1

A<int*, int*, 2> a5; // 歧义:
// 同时匹配 #3 和 #5
```

函数模板的重载

现在，来看看函数模板的重载。它和特殊化不是一回事，但与之有关。

C++允许对函数进行重载，同时还保证调用正确的函数：

```
int f( int );
long f( double );

int i;
double d;

f( i );    // 调用 f(int)
f( d );    // 调用 f(double)
```

同样，你也可以对函数模板进行重载，这给我们带来了最后一个问题：

3. 请看下面的声明：

```
template<typename T1, typename T2>
void g( T1, T2 );                                // 1
template<typename T> void g( T );                  // 2
template<typename T> void g( T, T );                // 3
template<typename T> void g( T* );                 // 4
template<typename T> void g( T*, T );               // 5
template<typename T> void g( T, T* );               // 6
template<typename T> void g( int, T* );              // 7
template<> void g<int>( int );                     // 8
void g( int, double );                            // 9
void g( int );                                    // 10
```

首先，让我们把问题简化一下，请注意，这里有两组被重载的 `g()`：其中一组接受一个参数，另外一组接受两个参数。

```
template<typename T1, typename T2>
void g( T1, T2 );                                // 1
template<typename T> void g( T, T );                // 3
template<typename T> void g( T*, T );               // 5
template<typename T> void g( T, T* );               // 6
template<typename T> void g( int, T* );              // 7
void g( int, double );                           // 9

template<typename T> void g( T );                  // 2
template<typename T> void g( T* );                 // 4
template<> void g<int>( int );                     // 8
void g( int );                                    // 10
```

注意，为了避免把水搅得太浑，在这里，我故意没有包含带两个参数且第二个参数有缺省值的重载函数。如果有这样一个函数，为了确定正确的调用次序，我们就得

考虑两个参数列表——一个是单参数函数（使用缺省值），一个是双参数函数（不使用缺省值）。

现在，让我们依次看看每一个调用：

以下每条语句分别调用上面的哪个函数？正确识别出合适的模板参数类型。

```
int          i;
double       d;
float        f;
complex<double> c;

g( i );      // a
```

A. 这将调用#10，因为它和#10 完全匹配，而且，这种非模板函数总是优先于模板函数（参见 C++ 标准第 13.3.3 节）。

```
g<int>( i );    // b
```

B. 这将调用#8，因为 `g<int>` 是被显式请求调用的。

```
g( i, i );    // c
```

C. 这将调用#3（T 为 int），因为这是最佳匹配。

```
g( c );      // d
```

D. 这将调用#2（T 为 `complex<double>`），因为没有其它的 `g()` 可以与之匹配。

```
g( i, f );    // e
```

E. 这将调用#1（T1 为 int, T2 为 float）。

你会认为#9 很接近——确实如此——但非模板函数只有在完全匹配的时候才会被优先选用。在这个例子中，#9 只是接近，而不是完全匹配。

```
g( i, d );    // f
```

F. 这个才会调用#9，因为它和#9 完全匹配，而且非模板函数会被优先选用。

```
g( c, &c );    // g
```

G. 这将调用#6（T 为 `complex<double>`），因为#6 是最接近的重载。模板#6 提供了 `g()` 的一个重载，其第二个参数是一个指针，指向的类型和第一个参数相同。

```
g( i, &d );    // h
```

H. 这将调用#7（T 为 double），因为#7 是最接近的重载。

```
g( &d, d );    // i
```

I. 这将调用#5 (T 为 double)。#5 提供了 g() 的一个重载，其第一个参数是一个指针，指向的类型和第二个参数相同。

只剩几个例子了，请继续往下看：

```
g( &d );      // j
```

J. 很明显（有了前面的分析做基础），这里请求的是#4 (T 为 double)。

```
g( d, &i );    // k
```

K. 另外几个重载也很接近，但只有#1 更出类拔萃 (T1 为 double, T2 为 int*)。

最后：

```
g( &i, &i );    // l
```

L. 这将调用#3 (T 为 int*)；虽然还有另外几个重载显式地声明了指针参数，但#3 最为接近。

这里有一条好消息：新型编译器一般都为模板提供了良好的支持，所以和过去相比，如今你可以更可靠地利用上面那样的特性，而且更具可移植性。

还（可能）有一条坏消息：如果你对以上问题全部回答正确，在对这些规则的理解方面，你可能比你现在使用的编译器还要强。

条款 11：Mastermind

难度：8

我们以一个智力游戏题结束本章，它展示了一些轻松有趣的编程技巧，同时让你体验如何构造自己的函数对象，以及如何复用标准库内建的算法和工具。

写一个简化的 Mastermind 游戏程序，只能用标准库容器、算法和流来实现。本书的基本目的是为了阐述严谨、有效的软件工程设计方法。但这一次，仅仅为了趣味性，我们的挑战多了一些轻松的意味。尽量少用 if、while、for 和其它关键字，尽量让程序包含最少的语句。对于本条款，即使代码晦涩也无所谓。

简化规则概述

游戏开始时，程序随机选取 4 个按某种内部顺序排列的棋子。每个棋子可以是三种颜色中的一种：红(R)、绿(G)或蓝(B)。例如，程序可能会挑选“RRBB”、“GGGG”或“BGRG”。

玩家连续猜测棋子的颜色顺序，直至猜测正确。每一次猜测中，程序告诉玩家两个数字。第一个数字是具有正确颜色的棋子的数目，不考虑顺序；第二个数字是颜色正确

并且位置也正确的棋子的数目。

下面是游戏进行过程的一个示例，其中，程序选择的颜色组合是“RRBB”：

```
猜测--> RBRR
3 1
猜测--> RBGG
2 1
猜测--> BBRR
4 0
猜测--> RRBB
4 4 - 成功!
```

解答

这里提供的两个方案不是仅有的正确答案。它们都可以很容易地扩充为更多的颜色和棋子，因为我们没有将棋子的颜色和组合限制在算法中。只要修改 colors 字符串，棋子颜色就可以扩充；修改 comb 字符串的长度，颜色组合的长度就可以直接改变。不足的是，这两个方案都没有做足够的出错检查。

这个问题中的一个要求是，使程序的语句数量最少；所以，只要有可能，这两个方案都会用逗号作为语句分隔符，而不用分号。在 C 或 C++ 代码中看到这么多的逗号，对大多数人来说都不太习惯；但实际上，可以使用逗号的地方比你原本想象的要多。另一个要求是，避免使用内建的控制关键字；为达此目的，我们将用三元运算符?: 替代 if/else。

方案 1

第一个方案的思想是，在处理每一次猜测时，对玩家输入的字符串只做一次遍历。第一个方案还是使用了一个 while。在第二个方案中，我们将用 find_if 来取代它（虽然在一定程度上损失了代码的清晰性）。

为了了解代码的基本结构，让我们从主程序开始分析：

```
typedef map<int, int> M;

int main() {
    const string colors("BGR"); // 可能的颜色
    string comb(4, '.'),           // 组合
           guess;                  // 当前的猜测
```

如果想扩充这个程序，使之可以处理更多的颜色，或者想让每个组合有更多的棋子，做法很简单：要想改变颜色，修改 colors。要想让猜测值更长或更短，就让 comb 更长或更短。

```
int cok, pok = 0;           // 正确的颜色和位置
M cm, gm;                 // 辅助结构
```

我们需要一个字符串，用以保存玩家想要找到的组合值；还需要另一个字符串，用以保存玩家的实际猜测值。除此之外是处理每一次猜测时用到的中间数据。

需要做的第一件事是生成组合值：

```
 srand( time(0) ),
 generate( comb.begin(), comb.end(),
 ChoosePeg( colors ) );
```

为内建的随机数生成函数（random number generator）设置开始点（seed）后，我们使用标准库的 generate() 产生一个组合值。这里，generate() 使用了一个类型为 ChoosePeg 的函数对象，用以生成组合值；这是通过调用 ChoosePeg 的函数调用运算符 operator()() 来实现的，对每一个棋子位置，它都会被调用一次，以确定那个位置的颜色。在后面，我们将看到 ChoosePeg 是如何运作的。

现在，我们需要的是一个主循环，用以输出提示信息并对猜测进行处理：

```
while( pok < comb.size() )
cout << "\n\nguess--> ",
cin >> guess,
guess.resize( comb.size(), ' ' ),
```

我们对输入错误做了一些很基本的处理。如果玩家的猜测值太长，那个猜测值就会被截短；如果太短，则会添上空格。为过短的猜测值添加空格给了玩家作弊的可能，玩家可以在猜测时只输入一个字母并不断尝试，直到找到那个正确的字母，然后，他（她）再不断输入两个字母的猜测值，从而找到第二个正确的字母，如此下去，直至猜出整个答案。这段代码不必考虑去防止这一情况。

下一步，我们需要清除工作区并处理这个猜测：

```
cm = gm = M();
```

我们必须通过两种方式来处理这个猜测：首先，我们要确定颜色正确且位置也正确的棋子数（pok，或“place okay”）。另外，我们还要确定颜色正确但位置不一定正确的棋子数（cok，或“color okay”）。为避免对猜测值遍历两次，我们首先使用 transform() 来处理这两个区间，让函数对象 CountPlace 去收集“每一种颜色有多少棋子出现在组合值和猜测值中”的信息：

```
transform( comb.begin(), comb.end(),
          guess.begin(), guess.begin(),
          CountPlace( cm, gm, pok ) ),
```

标准库算法 `transform()` 在两个输入区间上进行操作——这里，也就是组合值字符串和猜测值字符串，它们被看作字符序列。不过，`transform()` 的实际功能超过了我们的需要，它还可以产生输出，但我们不关心输出。忽略输出的最简单方法是：首先，确保所提供的 `CountPlace` 函数对象不修改任何东西，然后，将输出放回到猜测值字符串中；因而，这实际上是个空操作。

注意，函数对象 `CountPlace` 的主要目的是计算 `pok`，但它还对 `cm` 和 `gm` 这两个 `map` 进行了填充。然后，这些辅助性的 `map` 被用于第二个阶段。在第二个阶段，`for_each` 对颜色值（不是猜测值）进行循环，计算每种颜色有多少棋子相匹配：

```
for_each( colors.begin(), colors.end(),
          CountColor( cm, gm, cok ) ),
```

`for_each()` 算法可能是标准库中最著名的算法。它的功能是遍历一个区间，并将所提供的函数对象作用在区间的每个元素身上。在这里，`CountColor` 函数对象负责计算和统计：每种颜色有多少棋子在两个字符串中匹配。

完成了对猜测的所有处理工作后，我们输出结果：

```
cout << cok << ' ' << pok;
```

注意，`while` 循环体外不需要加上大括号，因为整个循环体是一个语句，真的要感谢那些逗号！

最后，当循环结束时，任务完成：

```
cout << " - solved!\n";
}
```

看过了整个主程序如何工作，我们再来看看三个辅助函数对象。

ChoosePeg

最简单的辅助函数对象是 `ChoosePeg`，它用于生成组合值：

```
class ChoosePeg
{
public:
    ChoosePeg( const string& colors )
        : colors_(colors) { }

    char operator()() const
    { return colors_[rand() % colors_.size()]; }
```

```
private:
    const string& colors_;
};
```

对 `operator()()` 的每一次调用生成一个棋子颜色。

如果不是因为 `ChoosePeg` 有一个指向合法颜色字符串的引用，它本来可以是一个普通函数；我们就会让 `colors` 成为一个全局字符串，从而污染全局名字空间。这个方案没有让 `colors` 字符串成为全局变量，没有让 `ChoosePeg` 依赖于全局数据；它避免了对全局数据的使用，从而使 `ChoosePeg` 得到更好的封装。

CountPlace

下一个 `CountPlace`，它负责计算 `pok`——颜色正确且位置也正确的棋子数。同时，它还负责累积有关猜测值和组合值字符串的统计信息，以供后面的 `CountColor` 在计算中使用。

```
class CountPlace
{
public:
    CountPlace( M& cm, M& gm, int& pok )
        : cm_(cm), gm_(gm), pok_(pok=0) { }

    char operator()( char c, char g )
    {
        return ++cm_[c],
               ++gm_[g],
               pok_ += (c == g),
               g;
    }

private:
    M &cm_, &gm_;
    int& pok_;
};
```

对 `operator()()` 的每一个调用会将“组合值中的一个棋子 `c`”和“猜测值中的相应棋子 `g`”进行比较。如果它们相等，我们递增 `pok_`。现在，这些 map 的用途更清楚了。每个 map 存储的是出现在给定字符串中的某一颜色（颜色的 `char` 值被转换为 `int`）的棋子数。

根据 `transform()` 的语义，`operator()()` 必须返回某个 `char`。返回什么其实无关紧要，因为，即使输出被返回到 `guess` 字符串中，`guess` 字符串也不会再被使用。为了代码的简明和整洁，我们返回猜测值的棋子颜色，从而使 `guess` 字符串保持不变。

CountColor

最后是 CountColor，它利用 CountPlace 产生的统计信息计算出 cok。它的做法很简单：对每一种可能的颜色，不管位置如何，它累计出相匹配的棋子的数目——也就是说，在组合值和猜测值中，那种颜色的棋子数目的最小值。

```
class CountColor {
public:
    CountColor( M& cm, M& gm, int& cok )
        : cm_(cm), gm_(gm), cok_(cok=0) { }

    void operator()( char c ) const
        { cok_ += min( cm_[c], gm_[c] ); }

private:
    M &cm_, &gm_;
    int& cok_;
};
```

总结

将以上结合起来，我们得到下面这个完整的程序：

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

typedef map<int, int> M;

class ChoosePeg
{
public:
    ChoosePeg( const string& colors )
        : colors_(colors) { }

    char operator()() const
        { return colors_[rand() % colors_.size()]; }

private:
    const string& colors_;
};

class CountPlace
{
public:
    CountPlace( M& cm, M& gm, int& pok )
        : cm_(cm), gm_(gm), pok_(pok=0) { }
```

```

char operator()( char c, char g )
{
    return ++cm_[c],
           ++gm_[g],
           pok_ += (c == g),
           g;
}

private:
    M &cm_, &gm_;
    int& pok_;
};

class CountColor {
public:
    CountColor( M& cm, M& gm, int& cok )
        : cm_(cm), gm_(gm), cok_(cok=0) { }

    void operator()( char c ) const
        { cok_ += min( cm_[c], gm_[c] ); }

private:
    M &cm_, &gm_;
    int& cok_;
};

int main() {
    const string colors("BGR"); // 可能的颜色
    string comb(4, '.'),           // 组合
           guess,                  // 当前的猜测
    int cok, pok = 0,              // 正确的颜色和位置
    M cm, gm;                    // 辅助结构

    srand( time(0) ),
    generate( comb.begin(), comb.end(),
              ChoosePeg( colors ) );

    while( pok < comb.size() )
        cout << "\n\nguess--> ",
        cin >> guess,
        guess.resize( comb.size(), ' ' ),
        cm = gm = M(),
        transform( comb.begin(), comb.end(),
                  guess.begin(), guess.begin(),
                  CountPlace( cm, gm, pok ) ),
        for_each( colors.begin(), colors.end(),
                  CountColor( cm, gm, cok ) ),
        cout << cok << ' ' << pok;
    cout << " - solved!\n";
}

```

方案 2

第二个方案的思想是，让主程序简单得不能再简单，它相当于只是说了一句话：“在输入中寻找组合值”。这里是主程序：

```
int main()
{
    srand( time(0) ),
    find_if( istream_iterator<string>(cin),
              istream_iterator<string>(),
              Combination() );
}
```

就这些！Combination 这个 predicate 则必须完成所有的工作。

Combination

显然，我们要在 Combination 的缺省构造函数中产生组合值：

```
class Combination
{
public:
    Combination()
        : comb_(4, '.')
    {
        generate( comb_.begin(), comb_.end(), ChoosePeg ),
        Prompt();
    }
}
```

注意，这个 ChoosePeg 和第一个方案中的 ChoosePeg 不同；这一点一会儿再做说明。生成组合值后，构造函数输出第一条提示信息。需要做的就这些。

下一个 is Combination::operator()()，它会对每个输入的猜测值字符串和组合值进行比较，如果匹配，会返回 true，否则返回 false：

```
bool operator()( string guess ) const // 一次
{
    int cok, pok; // 正确的颜色和位置

    return
        guess.resize( comb_.size(), ' ' ),
```

同样，我们做了一些基本的出错处理，如果需要，我们会调整 guess 的长度。

主要工作还是分两个步骤完成。不同的是，这一次两个步骤颠倒过来了；而且，我们不会作茧自缚，要求自己对输入只能做一次遍历。首先确定 cok——颜色正确但位置不一定正确的棋子数：

```
cok = accumulate( colors.begin(), colors.end(),
                  ColorMatch( 0, &comb_, &guess ),
                  ColorMatch::Count ),
```

过一会儿我们再来看 ColorMatch 这个辅助函数对象。

第二步是计算 pok——颜色和位置都正确的棋子数。为了完成这一功能，我们将使用一个看起来似乎不太合适的算法：

```
pok = inner_product( comb_.begin(), comb_.end(),
                      guess.begin(), 0,
                      plus<int>(),
                      equal_to<char>() ),
```

我们不是在这儿做数学上的矩阵运算，那么为什么还要使用 `inner_product()` 呢？简短的答案是：不用白不用。

详细一点的答案是：这个算法的行为和我们的需要最为接近。`inner_product()` 算法的行为是：

- 得到两个输入区间；
- 对这两个区间中排在首位的元素执行某一指定的操作（让我们称为 op2），然后对这两个区间中排在第二位的元素执行这一操作，等等；
- 执行另一个指定操作（让我们称为 op1），用以合并前面的计算结果。

`op1` 和 `op2` 的缺省操作刚好分别是加法和乘法。这和我们的需要只有一点点不同。对于 `op1`，加法没关系；只是对于 `op2`，我们希望，在两个字符相等的时候它返回 1，否则返回 0，这刚好和“用 `equal_to<char>` 来进行相等比较操作并将结果由 `bool` 提升为 `int`” 所得到的效果一样。

如果 `inner_product()` 的名字还是困扰着你，那么，可将这个标准算法想象成 `accumulate()` 和 `transform()` 的一种混合物。`accumulate()` 在单个输入区间上执行计数操作；`transform()` 处理两个输入区间，它提供了一点灵活性：可以就“如何对两个区间中的元素进行操作”进行定制。数学上的内积运算是以上情况的特例，但 `inner_product()` 算法更具一般性。正如我们已经看到的那样，它适用于对输入序列进行不同方式的合并和计算。

```
cout << cok << ' ' << pok,
(pok == comb_.size())
? (cout << " - solved!\n", true)
: (Prompt(), false);
}
```

最后，这个运算符还提供用户界面，包括猜测值的显示，以及后续的提示信息，或者程序结束信息。

至于非静态数据成员，我们只需要显而易见的那一个：

```
private:
    string comb_; // 实际组合
```

非静态成员到此为止。其余的都是 Combination 的静态成员：

```
static char ChoosePeg() { return colors[rand() % colors.size()]; }
```

注意，由于封装性发生了变化——colors 字符串和 ChoosePeg() 在同一空间——我们可以简化 ChoosePeg()，使之成为一个无状态函数。

```
static void Prompt() { cout << "\n\nguess--> "; }
static const string colors; // 可能的颜色
};

const string Combination::colors = "BGR";
```

如今，只有 Combination 类需要知道“可能的颜色”和“猜测值长度”，所以我们可以很好地将这些信息隐藏起来，即，将它们藏在 Combination 的空间范围内。

ColorMatch

我们还需要一个函数对象 ColorMatch。请记住，它得像下面这样使用：

```
cok = accumulate( colors.begin(), colors.end(),
                  ColorMatch( 0, &comb_, &guess ),
                  ColorMatch::Count );
```

这里有几个关键之处。请注意，ColorMatch 实际上是那种正在被累计的值。那么，这意味着 accumulate() 将返回最终的 ColorMatch 值，然而我们想找的是一个 int。这太容易了——我们只用提供一个到 int 的转换：

```
class ColorMatch {
public:
    ColorMatch( int i, const string* s1, const string* s2 )
        : cok_(i), s1_(s1), s2_(s2) { }

    operator int() const { return cok_; }
```

现在，当像前面那样使用 accumulate() 时，我们不是对 int 这种普通值进行累加，而是使用 ColorMatch 对象来执行 Count() 计算。也就是说，针对 colors 中的每一个 char，accumulate() 会执行 ColorMatch::Count() 函数，这个函数接受两个参数，一个是 ColorMatch 值，表示到目前为止的计数，另一个是将要处理的 colors 字符：

```

static ColorMatch Count( ColorMatch& cm, char c )
{
    return
    ColorMatch(
        cm.cok_ +
        min( count( cm.s1_->begin(), cm.s1_->end(), c ),
            count( cm.s2_->begin(), cm.s2_->end(), c ) ),
        cm.s1_, cm.s2_ );
}

```

Count() 的任务是，计算出有多少颜色为 c 的棋子同时在两个字符串中存在（不一定处于相同的位置），并将结果累加到目前的 ColorMatch 计数中。在完成这一任务时，我们只不过使用了另一个标准算法 count()，它确定出每个字符串中颜色为 c 的棋子数，然后将较小的值累加起来。于是，Count() 返回一个新的 ColorMatch 对象，这个对象包含新的计数。因为针对每一种可能的颜色，Count() 都会被调用一次，所以，最后的 ColorMatch() 对象的 cok_ 值正是我们想要的值：颜色正确但位置不一定正确的棋子数。

最后，是这个函数对象保留的一些私有状态值：

```

private:
    int cok_;
    const string *s1_, *s2_;
};

```

总结

再一次，将以上代码合并成完整程序后，我们可以更容易地看到不同部分之间如何相互配合与共存：

```

#include <iostream>
#include <algorithm>
#include <numeric>
#include <functional>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

class ColorMatch {
public:
    ColorMatch( int i, const string* s1, const string* s2 )
        : cok_(i), s1_(s1), s2_(s2) { }

    operator int() const { return cok_; }
}

```

```

static ColorMatch Count( ColorMatch& cm, char c )
{
    return
        ColorMatch(
            cm.cok_ +
            min( count( cm.s1_->begin(), cm.s1_->end(), c ),
                  count( cm.s2_->begin(), cm.s2_->end(), c ) ),
            cm.s1_, cm.s2_ );
}

private:
    int cok_;
    const string *s1_, *s2_;
};

class Combination
{
public:
    Combination()
        : comb_(4, '.')
    {
        generate( comb_.begin(), comb_.end(), ChoosePeg ),
        Prompt();
    }

    bool operator()( string guess ) const // 一次
    {
        int cok, pok; // 正确的颜色和位置

        return
            guess.resize( comb_.size(), ' ' ),
            cok = accumulate( colors.begin(), colors.end(),
                ColorMatch( 0, &comb_, &guess ),
                ColorMatch::Count ),
            pok = inner_product( comb_.begin(), comb_.end(),
                guess.begin(), 0,
                plus<int>(),
                equal_to<char>() ),
            cout << cok << ' ' << pok,
            (pok == comb_.size())
                ? (cout << " - solved!\n", true)
                : (Prompt(), false);
    }

private:
    string comb_; // 实际组合

    static char ChoosePeg() { return colors[rand() % colors.size()]; }
    static void Prompt() { cout << "\n\nguess--> "; }
    static const string colors; // 可能的颜色
};

```

```

const string Combination::colors = "BGR";

int main()
{
    srand( time(0) ),
    find_if( istream_iterator<string>(cin),
              istream_iterator<string>(),
              Combination() );
}

```

两种方案的比较

本条款有两个目的。一个目的是体验一些 C++ 特性所带来的乐趣，所以我们欣赏了一些具有创造性的逗号运算符使用方式；在实际产品代码中，这种使用方式即使出现过，也应该很少见。

另一个目的则更严肃，它是为了让你更好地熟悉泛型程序设计，熟悉 C++ 标准库必须提供、预装和准备就绪的一切。每个方案都通过不同的方式、朝着不同的目标达成这一目的。方案 1 的主要目标是避免对输入进行多次遍历。制定这一目标并不是出于性能上的考虑，因为在这个简单程序中，运行时期由用户输入支配，而非程序的处理过程；但是，避免多次遍历是一种很有用的技术，现实世界中的每一位程序员都应该了解它。什么时候应该避免对数据多次遍历？一个常见的例子是输入数据非常大的时候；如果数据比内存还大，需要从磁盘读取两次甚至多次，那么，多次遍历的开销就会大得惊人——很显然，这时候，避免对缓慢的辅助存储器的不必要访问就显得十分重要。另一个例子是不可能进行多次遍历的时候——你可能只有一次获取数据的机会，例如，当输入来自一个输入迭代器的时候。

方案 2 的主要目标是想拥有一个干练明了的主程序，以及更具封装性的内部代码。虽然方案 2 的代码量比方案 1 多一些，但它更清晰；方案 1 通过辅助结构在主循环的两个阶段之间建立间接的联系，因而可能让人感到晦涩难懂，而方案 2 避免了这一点。由于我们允许对输入数据进行两次遍历，方案 1 中的那些数据结构就不再需要，那两次遍历也变得泾渭分明，而不是相互牵连，因而，更容易让人理解。

两个方案都让我们体验了如何重复使用（reuse）标准库，也都演示了开发产品代码时十分有用的技术。（但是，在产品代码中，我会离那些逗号远远的；将它们用在这儿纯属娱乐。）

优化与性能

对程序员来说，效率总是很重要。在 C 和 C++ 的传统中，效率是重要支柱之一，“不要为没有使用的东西支付任何成本”这一指导原则——也称为零成本原则——总是语言设计和程序库设计的中心，而且，在很大程度上也确实得到了实现。

在本章以及书后的两个附录中，我们对一些重要的 C++ 优化问题进行了深入的观察，并分析了它们对现实世界代码的影响。应该在何时优化你的代码？如何优化？`inline` 到底做了些什么？为什么花哨的优化能够（而且确实会）让我们陷入麻烦？最后一点，并且对我来说最有趣的一点：如果你是在写多线程代码，上面一些问题的答案会如何发生变化？毕竟，我们关心的是现实世界中的效率问题；虽然 C++ 标准对线程避而不谈，但在程序设计领域的第一线，每天都有越来越多的程序员在写多线程的 C++ 代码。他们会关心这些问题的答案。

条款 12：内联

难度：4

将一个函数说明为 `inline` 意味着告诉编译器：编译器可以选择不生成函数的副本，而是直接在调用它的每一个地方都放置该函数的拷贝。这在某些情况下非常有用，但并不总能提高效率。如果在函数的实现部分包含大量的计算，那么将它声明为 `inline` 可能会降低效率，因为它会生成许多副本，从而消耗更多的内存空间。因此，只有在函数的实现部分非常简单时，将它声明为 `inline` 才能真正提高效率。

1. `inline` 有什么作用？
2. 将函数内联会提高效率吗？
3. 何时应当决定使用内联函数？如何决定？

1. `inline` 有什么作用？

将一个函数说明为 `inline` 意味着告诉编译器：编译器可以将这个函数代码的拷贝直接放在每一个使用这个函数的地方。编译器可以选择这么做，或不这么做；如果编译器确实这么做了，将会避免函数调用的发生。

2. 将函数内联会提高效率吗？

不一定。

首先，在回答这个问题之前，如果不先问问自己到底想优化“什么”，你就会落入一个著名的陷阱。第一个问题应该是：“你所说的效率指的是什么？”在上面的提问中，所谓的效率指的是程序体积吗？抑或是内存占用？执行时间？开发速度？编译时间？或是其它什么东西？

其次，和大多数人的看法相反：对于效率的各个方面，内联可能使之改善，也可能使之恶化：

(a) 程序体积。许多程序员认为，内联一定会增加程序的体积，因为程序拥有的将不只是函数代码的一份拷贝，编译器会在使用了那个函数的每个地方生成一份拷贝。通常来说这是对的，但并非总是如此。如果和“编译器为了执行函数调用而不得不生成其代码”的体积相比，内联函数的体积比它还小，那么，内联会减小程序体积。

(b) 内存占用。除了（上面所说的）基本程序体积外，内联通常对程序的内存使用没有影响，或极少有影响。

(c) 执行时间。很多程序员认为，将函数内联一定会提高运行速度，因为它避免了函数调用的开销；而且，透过了函数调用这层“障碍”，编译器的优化程序就有了更多大显身手的机会。这可能是正确的，但不总是正确：如果函数不是被极其频繁地调用，整个程序的执行时间通常不会有明显的改善。实际上，事情有可能适得其反。如果内联增加了调用函数 (calling function) 的体积，它会降低调用者的“引用局部性 (locality of reference)”（译注：参见[Meyers96]条款 18）；这意味着，如果调用者的内部指令循环 (inner loop) 不再和处理器高速缓存的大小相匹配，整个程序的执行速度实际上会降低。

不要忘记：客观地说，大多数程序的运行速度并非受限于 CPU。最常见的瓶颈可能在于 I/O 上的限制，这包括很多方面，如网络带宽或延迟、对文件或数据库的访问，等等。

(d) 开发速度和编译时间。为了得到最有效的利用，被内联的代码必须对调用者可见；这意味着，调用者必须依赖于被内联代码的内部细节。依赖另一个模块的内部实现细节必然增加模块的实际耦合性（但不会增加理论耦合性，因为调用者实际上没有使用被调用者的任何内部实现）。通常情况下，当普通函数被修改时，调用者无需重新编译，只需重新连接。当内联函数被修改时，调用者必须重新编译。还有，内联函数本身会在调试时期单独影响开发速度，因为，要想单步跟踪到内联函数的内部，或者在内联函数内部管理断点，对大多数调试器来说会更困难。

有一种情况下，一些人会认为内联是对开发速度的一种优化（这一点有一些争议）——为了避免让数据成员为公有成员，提供一个存取函数 (accessor，见后) 是好的

做法，但写这样一个存取函数的代价可能很高。这种情况下，一些人会认为，使用内联会带来好的编码风格和更好的模块独立性。

最后请记住，如果你想用什么方式提高效率，总是先借助你的算法和数据结构。它们会给你的程序带来数量级的整体改善，而内联之类的过程优化（process optimization）通常（注意，“通常”）收效甚微。

不妨说“现在不行”

3. 何时应当决定使用内联函数？如何决定？

和使用其它任何一种优化技术一样，答案是：在分析工具告诉你这样做之前，不要贸然行事。这条原则有几个合理的例外——在有些场合下你可以毫不迟疑地内联一个函数：例如空函数，而且会持续保持为空；或者，你非得这么做不可的时候——例如，在写一个非输出模板（non-exported template）的时候。

设计准则

使用优化的第一条原则：不要使用它。

使用优化的第二条原则：还是不要使用它。

结论：只要增加了耦合性，内联就总是会带来成本；绝对不要为某个东西事先支付成本，除非你知道它会带来好处——也就是说，回报大于支出。

“但我总能找到瓶颈在哪儿！”你会这样想。别着急，不是你一个人这样想。大多数程序员都或多或少地这样想过，但他们还是错了。完全错了。对于代码中真正的瓶颈所在，程序员是臭名昭著的猜测者。有时侯，我们撞大运似地蒙对了。但大多数时候，我们的猜测是错误的。

通常，只有实验数据（或称分析结果）可以帮助我们找出真正的热点所在。如果不借助某种分析工具，十有八九，一个程序员不可能识别出他（她）的代码中头号热点或瓶颈所在。干这行很多年了，我曾遇到过几个反对这一事实的程序员，他们（或他们的同事）坚持认为这一事实对他们不适用，声称自己一向能够“感觉到”自己代码中的热点所在。多年来，我从没有看到过这样的宣言变成一贯的事实。我们善于欺骗自己。

最后注意，使用这条准则的另一个实际原因是：对于哪一个内联函数不应该被内联，分析工具并不善于识别。

关于密集计算任务（例如数值计算程序库）

一些人要编写短小紧凑的程序库代码，例如高级科学和工程计算程序库，这些人有时可以凭直觉使用内联，并做得很不错。然而，即使是这些程序员，他们也倾向于明智地使用内联，认为优化宜迟不宜早。注意，写一个模块，然后通过“打开内联”和“关闭内联”的方式来比较性能，这通常不是一个好主意，因为“全开”和“全闭”是一种很粗糙的分析方法，它只能告诉你一般情况。它不会告诉你哪个函数受益，也不会告诉你每个函数受益多少。即使在这些情况下，你也应该使用分析工具并基于它的建议去优化。

关于存取函数

会有一些人争辩说：只有一行代码的存取函数（例如“`X& Y::f() { return myX_; }`”）是个合理的例外，它“可以”或“应该”被自动内联。我知道这样做的道理，但要小心行事。不管怎么说，所有被内联的代码都会增加耦合性。所以，除非你事先确信内联会带来好处，否则，将使用内联的决定延迟到分析之后是没有坏处的。到了那时，如果分析工具真的指出内联会有好处，你至少知道，你正在做的是值得做的事；而且，你也将耦合和可能的编译开销延迟了——延迟到你确实知道内联真的有必要时候。这样做是不会让你吃亏的，真的！

设计准则

在性能分析证明确实必要之前，避免内联或详细优化。

条款 13：缓式优化，之一：

一个普通的旧式 String

难度：2

`Copy-on-write`（写入时拷贝，也称“`lazy copy`”（懒式拷贝））是一种常用优化。它运用了引用计数技术。你知道如何实现它吗？在这个系列的第一个条款中，我们来分析一个简单而普通的 String 类。它完全没有使用引用计数技术。在本短系列的其余部分，我们将看到，增加 `copy-on-write` 语义会对这个类产生怎样的影响。

请分析下面这个简化的 String 类。（注意：这个 String 类并不能作为一个完善的字符串使用。它经过了简化，只是用作一个示例；它省略了作为一个真正的字符串类的很多常用操作。特别是，我没有提供 `operator=()`，因为它的实现实质上和拷贝构造函数相同）

```

namespace Original
{
    class String
    {
    public:
        String();           // 开始为空
        ~String();          // 释放缓冲区
        String( const String& ); // 进行完整拷贝
        void Append( char ); // 添加一个字符

        // ... 省略 operator=( ) 等 ...
    private:
        char*   buf_;           // 分配的缓冲区
        size_t  len_;           // 缓冲区长度
        size_t  used_;          // 实际使用的字符数
    };
}

```

这是一个简单的 String，不包含任何特别的优化。当你拷贝一个 Original::String 时，新对象立即分配自己的缓冲区，你立即有了两个完全独立的对象。

你的任务是：实现 Original::String。

解答

实现 Original::String

这里是一个直截了当的实现。缺省构造函数和析构函数写起来很简单：

```

namespace Original {

    String::String() : buf_(0), len_(0), used_(0) { }

    String::~String() { delete[] buf_; }
}

```

接着，为了实现拷贝构造函数，我们只需要分配一个新的缓冲区，并使用标准算法 copy() 来创建一个原始内容的拷贝：

```

String::String( const String& other )
: buf_(new char[other.len_]),
len_(other.len_),
used_(other.used_)

```

```

    {
        copy( other.buf_, other.buf_ + used_, buf_ );
    }

```

为了让代码更清晰，我另外写了一个 Reserve() 辅助函数，因为除了 Append()，其它修改操作函数（mutators）也需要用到它。Reserve() 保证我们的内部缓冲区的长度至少有 n 字节，它使用指数增长策略来分配需要追加的空间：

```

void String::Reserve( size_t n )
{
    if( len_ < n )
    {
        size_t newlen = max( len_ * 1.5, n );
        char* newbuf = new char[ newlen ];
        copy( buf_, buf_+used_, newbuf );
        delete[] buf_; // 现在，所有实际工作已经完成。
        buf_ = newbuf; // 所以，获得所有权
        len_ = newlen;
    }
}

```

最后，Append() 在确认有足够的空间后，添加新的字符：

```

void String::Append( char c )
{
    Reserve( used_+1 );
    buf_[used_++] = c;
}

```

附笔：最佳缓冲区增长策略是什么？

当 String 对象当前已分配的缓冲区不够用时，它需要分配一个更大的缓冲区。问题的关键是：新缓冲区该分配多大才好？请注意，这个问题很重要，因为我们在下面所做的分析也同样适用于其它使用了缓冲区的结构和容器，如标准的 vector。

这里介绍了几个普遍运用的缓冲区增长策略。在表示每一个策略的复杂度时，我用到了两个指标，即，对于一个最终长度为 N 的字符串，它需要分配的次数和一个给定的字符必须被拷贝的平均次数。

(a) 精确增长 (Exact growth)。在这一策略中，新缓冲区大小和当前操作所要求的大小完全一样。例如，添加一个字符后又添加另一个字符，这一定会造成两次分配操作。首先，一个新缓冲区会被分配，用以存储现有字符串和第一个新字符。接着，将分配另一个新缓冲区，用以存储前次分配的字符串和第二个新字符。

- 优点：不浪费空间
 - 缺点：性能差。这个策略需要 $O(N)$ 次分配，每个 `char` 需要平均 $O(N)$ 次拷贝操作，但最坏情况下的常数因子很高（在下面的策略 b) 中，如果其增长量为 1，那么这两个策略一样）。常数因子由用户代码控制，而不是 `String` 的实现者。
- (b) 固定增量增长 (Fixed-increment growth)。新缓冲区总是比当前缓冲区增大一个固定值。例如，64 字节的增量意味着所有字符串缓冲区的长度为 64 字节的整数倍。
- 优点：空间浪费少。缓冲区中未使用的空间量受增量大小限制，不会随字符串长度发生变化。
 - 缺点：性能一般。这个策略需要 $O(N)$ 次分配，每个 `char` 需要平均 $O(N)$ 次拷贝操作。也就是说，“分配次数”和“一个给定的 `char` 被拷贝的平均次数”随着字符串的长度线性变化。然而，常数因子的控制权掌握在 `String` 的实现者手中。
- (c) 指数增长 (Exponential growth)。新缓冲区比当前缓冲区大 F 倍。例如，如果 $F = .5$ ，那么，在向一个已经满载的 100 字节字符串添加一个字符时，将分配一个长度为 150 字节的缓冲区。
- 优点：性能上佳。这个策略需要 $O(\log N)$ 次分配，每个 `char` 平均需要 $O(1)$ 次拷贝操作。也就是说，分配次数随字符串长度成对数 (\log) 级变化，但一个给定字符被拷贝的平均次数是一个常数，这意味着，较之“事先已经知道字符串长度的情况下所需要的拷贝工作量”，使用这一策略来产生一个字符串所需要的拷贝工作量至多只比前者多一个常数倍。
 - 缺点：有些浪费空间。缓冲区中未使用的空间量一定总是少于 $N \cdot F$ 字节，但平均还是有 $O(N)$ 级空间浪费。

下表总结了各个策略的利弊：

增长策略	分配	字符拷贝	空间浪费
精确增长	$O(N)$	$O(N)$	无
	高常数	高常数	
固定增量增长	$O(N)$	$O(N)$	$O(1)$
指数增长	$O(\log N)$	$O(1)$	$O(N)$

通常来说，指数增长策略表现最佳。假设一个字符串开始为空，然后增长到 1200 个字符的长度，固定增量增长需要 $O(N)$ 次分配，平均需要对每个字符拷贝 $O(N)$ 次（这

样，如果使用 32 字节增量，将需要 38 次分配，每个字符平均需要 19 次拷贝）；指数增长需要 $O(\log N)$ 次分配，平均只需要为每个字符做 $O(1)$ 次——1 次或 2 次——拷贝（是的，这是真的；参见下面的数据）；这样，如果使用的倍数为 1.5，将需要 10 次分配，每个字符平均需要 2 次拷贝。

	1 200 个 char 的字符串		12 000 个 char 的字符串	
	固定增量增长 (32 字节)	指数增长 (1.5x)	固定增量增长 (32 字节)	指数增长 (1.5x)
内存	38	10	380	16
分配次数				
每个 char	19	2	190	2
平均拷贝次数				

结果令人吃惊。更多信息请参阅 Andrew Koenig 在 JOOP (Journal of Object-Oriented Programming) 1998 年第九期的专栏。在那儿，Koenig 还说明了为什么在通常情况下最佳增长因子不是 2，而可能是 1.5 左右。他也说明了为什么一个给定字符的平均拷贝次数是常量——也就是说，不依赖于字符串的长度。

条款 14：缓式优化，之二：

引入缓式优化

难度：3

Copy-on-write 是一项有用的缓冲技术。你知道如何实现它吗？

Original::String (见条款 13) 好倒是好，但有时候，在得到一个字符串对象的拷贝后，用户可能不会在使用中对它做任何修改，然后又把它丢弃了。

“这好像有点丢人，”条款 13 中 Original::String 的设计者可能会对自己不满，“我每次都做了分配新缓冲区的所有工作（开销会很昂贵），可是，如果所有用户只是从新字符串中读取数据然后将其摧毁，那么，我所做的其实完全没有必要。我可以只是让两个字符串对象在底层共享一个缓冲区，暂时避免拷贝操作；只是在确实知道需要拷贝的时候，也就是，当其中一个对象试图修改这个字符串的时候，我才进行拷贝。采用这种方法，如果用户永远不修改这个拷贝，我就永远不用做额外的工作！”

脸上带着微笑，眼里充满着坚定，这个程序员设计了一个 Optimized::String，它使用了一个 copy-on-write 实现（也称“缓式拷贝”），对底层字符串实体实施引用计数：

```

namespace Optimized
{
    class StringBuf
    {
public:
    StringBuf();           // 开始为空
    ~StringBuf();          // 删除缓冲区
    void Reserve( size_t n ); // 保证 len >= n

    char*     buf;          // 分配的缓冲区
    size_t    len;           // 缓冲区长度
    size_t    used;          // 实际使用的字符数量
    unsigned refs;          // 引用计数

private:
    // 禁止拷贝...
    //
    StringBuf( const StringBuf& );
    StringBuf& operator=( const StringBuf& );
};

class String
{
public:
    String();           // 开始为空
    ~String();          // 递减引用计数
    // (如果 refs==0 则删除缓冲区)
    String( const String& ); // 指向同一缓冲区,
    // 并递增引用计数
    void Append( char ); // 增添一个字符

    // ...省略 operator=( ) 等...

private:
    StringBuf* data_;
};
}

```

你的任务是：实现 Optimized::StringBuf 和 Optimized::String。^⑩ 你可能需要增加一个私有的 String::AboutToModify() 辅助函数，以简化处理逻辑。

^⑩ 本条款附带演示了名字空间的另一种用途——使得表达更清晰。它使得代码不仅在阅读的时候更容易更自然，而且在用人类语言交流起来时也更容易更自然。上面的做法比分别写两个名为 OriginalString 和 OptimizedString 的类要好得多，更不用担心那些冗长的名称会给示例代码的阅读和书写带来某些困难。明智地使用名字空间还可以增强产品代码的可读性，以及设计研讨会和代码阅读中的“可交流性”。

解答

首先看看 `StringBuf`。注意，缺省的以成员为单位的拷贝和赋值操作对于 `StringBuf` 来说没有意义，所以二者都被禁止掉了（声明为 `private` 但没有定义）。

这里，`Optimized::StringBuf` 所完成的工作正是当初 `Original::String` 所做工作的一部分。`StringBuf` 的缺省构造函数和析构函数正如你所估计的那样：

```
namespace Optimized
{
    StringBuf::StringBuf() : buf(0), len(0), used(0), refs(1) { }

    StringBuf::~StringBuf() { delete[] buf; }
```

`Reserve()` 也和 `Original::String` 中的那个类似：

```
void StringBuf::Reserve( size_t n )
{
    if( len < n )
    {
        size_t newlen = max( len * 1.5, n );
        char* newbuf = new char[ newlen ];
        copy( buf, buf+used, newbuf );

        delete[] buf; // 现在，所有实际工作已经完成，
        buf = newbuf; // 所以，获得拥有权
        len = newlen;
    }
}
```

`StringBuf` 就这些。

再看看 `String` 本身。缺省构造函数很容易实现：

```
String::String() : data_(new StringBuf) { }
```

在析构函数中，我们要记得对引用计数进行管理，因为可能有其它的 `String` 对象正在共享相同的 `StringBuf` 实体。如果其它 `String` 对象还在使用这个 `StringBuf`，我们就无需理会它，只用递减引用计数以表明我们不再需要它，然后退出。但如果我们将是 `StringBuf` 的最后一个客户，我们就要清除它：

```

String::~String()
{
    if( --data_>refs < 1 ) // 最后一个
    {
        delete data_; // ...关灯（删除缓冲区）
    }
}

```

仅有的另外一个需要修改引用计数的地方是拷贝构造函数，在那儿，为了实现“缓式拷贝”语义，我们只需简单地指向另一个 String 已有的 StringBuffer，并递增计数以记录我们的存在。这是一种“浅拷贝”。只是在需要对“共享这个缓冲区的其中一个字符串”进行修改时，我们才会分离这个实体（即，执行“深拷贝”）：

```

String::String( const String& other )
: data_(other.data_)
{
    ++data_>refs;
}

```

为了提高代码的清晰性，我还另外实现了一个辅助函数 AboutToModify()；因为除了 Append()，其它修改操作函数（mutators）也需要用到它。AboutToModify()确保我们拥有一个非共享的内部缓冲区拷贝——如果在此之前没有执行深拷贝，现在就会执行。为了方便，AboutToModify()还提供了一个参数，表示欲分配缓冲区的最小值，这样，我们就不会不必要地得到一个已经满载的字符串拷贝，然后又立即转过头去执行第二次分配以获得更大的空间。

```

void String::AboutToModify( size_t n )
{
    if( data_>refs > 1 )
    {
        auto_ptr<StringBuf> newdata( new StringBuffer );
        newdata->Reserve( max( data_->len, n ) );
        copy( data_>buf, data_>buf+data_>used,
newdata.get()->buf );
        newdata->used = data_->used;

        --data_>refs; // 现在，所有实际工作已经完成，
        data_ = newdata.release(); // 所以，获得所有权
    }
    else
    {
        data_>Reserve( n );
    }
}

```

既然其它部分已经就绪，Append()就很简单了。和前面一样，它只是宣告它将要修改字符串，从而保证物理字符串缓冲区不被共享，而且其大小足以容纳另外一个字符。

然后，它继续前进，执行修改操作：

```
void String::Append( char c ) {
    AboutToModify( data_->used+1 );
    data_->buf[data_->used++] = c;
}

}
```

所有的工作就这么多。下一步，我们将使接口更丰富一些，并看看这会带来什么变化。

条款 15：缓式优化，之三：

迭代器与引用

难度：6

在“字符串类的第二部分”，我们来看看，指向 copy-on-write 字符串的引用和迭代器，具有怎样的效果。你能发现问题所在吗？

请看下面这个运用了 copy-on-write 技术的 Optimized::String 类，它来自条款 14，但增加了两个新的函数：Length() 和 operator[]().

```
namespace Optimized
{
    class StringBuffer
    {
        public:
            StringBuffer();                                // 开始为空
            ~StringBuffer();                               // 删除缓冲区
            void Reserve( size_t n ); // 保证 len >= n

            char*     buf;                                // 分配的缓冲区
            size_t    len;                                // 缓冲区长度
            size_t    used;                               // 实际使用的字符数
            unsigned refs;                             // 引用计数

        private:
            // 禁止拷贝
            //
            StringBuffer( const String& );
            StringBuffer& operator( const StringBuffer& );
    };

    class String
    {
        public:
            String();                                    // 开始为空
```

```

~String();           // 递减引用计数
                    // (如果 refs==0, 删除缓冲区)
String( const String& ); // 指向同一缓冲区,
                        // 并递增引用计数
void Append( char );   // 添加一个字符

size_t Length() const; // 字符串长度

char& operator[](size_t); // 元素访问
const char operator[](size_t) const;

// ...省略 operator=(...) 等...

private:
    void AboutToModify( size_t n );
                    // 缓式拷贝, 保证 len>=n
    StringBuf* data_;
};

}

```

这样，就可以写出下面的代码：

```

if( s.Length() > 0 )
{
    cout << s[0];
    s[0] = 'a';
}

```

实现 Optimized::String 中的新成员函数。增加了新成员函数后，有其它什么成员需要修改吗？给出理由。

解答

请看下面这个运用了 **copy-on-write** 技术的 Optimized::String 类，它来自条款 14，但增加了两个新的函数：**Length()** 和 **operator[]()**。

本条款的主旨在于说明，为什么增加 **operator[]()** 这样的功能会改变 Optimized::String 的语义，并且足以影响到类的其它部分。但，还是从头开始说起吧！

实现 Optimized::String 中的新成员函数：

Length() 函数很简单：

```

namespace Optimized
{
    size_t String::Length() const
    {
        return data_->used;
    }
}

```

然而，operator[]就不像表面上那么简单了。在下面的讨论中，我想请你注意的是，operator[]所做的（对非 const 版本来说，它返回一个引用，指向字符串的内部）实际上和 begin()、end()为标准 string 所做的（它们返回迭代器，“指向” string 的内部）没有什么不同。我们在下面所讨论的一些事项，任何 std::basic_string 的 copy-on-write 实现同样会碰上。

为可共享的 String 提供 operator[]

这是第一次幼稚的尝试：

```
// 低劣的方案：对于 operator[]的第一个幼稚的尝试
//
char& String::operator[]( size_t n )
{
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    return data_->buf[n];
}
```

这个设计很低劣，它甚至经不起随便一个测试。请看：

```
// 例 15-1：为什么第 1 个尝试不可行
//
void f( const Optimized::String& s )
{
    Optimized::String s2( s );      // 获取这个字符串的一个拷贝
    s2[0] = 'x';                  // 哎呀：也改动了 s!
}
```

你可能在想，写出例 15-1 的那个可怜的程序员应该会对这个副作用有点恼火。你的想法没错。毕竟，这个程序员本来只想改变一个字符串，但两个字符串都同时发生了改变；而且，那个被隐蔽地改变的字符串实际上应该为 const！这样，那个可怜的 f() 的设计者就有点像个骗子，因为他曾经承诺某个东西为 const，但最后又无意中改变了它。但他没有想说谎，因为代码中丝毫没有类型转换，也没有其它破坏性或反常的操作。但无论如何，如果事情发生了严重错误，那绝不是个好看的场面，除非你属于那种喜欢看汽车残骸的古怪人。

所以，让我们回到 String::operator[]，继续进行尝试。最低程度上，我们得保证字符串不被共享；否则，调用者会在无意中改动在他看来毫无联系的两个字符串。“啊哈，”

那个曾经幼稚过一次的程序员会想，“我可以调用 `AboutToModify()`，从而保证我所使用的是一个非共享的实体。”于是，他写下了下面的代码：

```
// 低劣的方案：对于 operator[] 的第二个尝试，但还是不够
//
char& String::operator[]( size_t n )
{
    AboutToModify( data_->len );
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    // 这一次不需要检查共享状态
    return data_->buf[n];
}
```

这看起来不错，但还是不够。问题在于，只需稍稍变化一下例 15-1 中的那段测试代码，我们就又回到了和以前相同的问题上：

```
// 例 15-2：为什么第 2 个尝试也不正确
//
void f( Optimized::String& s )
{
    char& rc = s[0]; // 获取第一个 char 的引用
    Optimized::String s2( s ); // 获取这个字符串的一个拷贝
    rc = 'x';           // 啊呀：也改动了 s2!
}
```

你可能在想，写出例 15-2 的那个可怜的程序员也应该会对这个意外感到心烦意乱。你的想法还是没错，但直到我写作本书之时，在某些很普及的 `basic_string` 实现中，这个有关 `copy-on-write` 的臭虫还确实存在。

问题在于，引用是在原始字符串没被共享的时候获取的，但随后这个字符串又被共享，于是，通过引用，单个的修改操作同时改动了两个 `String` 对象的可见状态。

重要概念：“不可共享”的 `String`

当非 `const` 版本的 `operator[]()` 被调用时，我们除了得到 `StringBuf` 的一个非共享的拷贝外，还需要将这个字符串标记为“不可共享”，以应付用户想起这个引用（并随后去使用它）的情况。

那么，把这个字符串标记为“永远不可共享”就可以了，但这样做有点过头。其实，我们需要做的只不过是将这个字符串标记为“暂时不可共享”。要明白我的意思，

请认识到这么一点：如果 `operator[]` 返回了一个指向字符串内部的引用，那么，在下一次修改操作之后，我们必须认为那个引用是无效的。也就是说，假如有下面一段代码：

```
// 例 15-3: 为什么修改操作
// 会使引用失效
//
void f( Optimized::String& s )
{
    char& rc = s[0];
    s.Append( 'i' );
    rc = 'x';      // 错误: 如果 s 执行了重新分配,
}                      // 缓冲区可能已经被移动
```

这段代码早就该被注明是不合法的——无论这个字符串是否使用了 `copy-on-write` 技术。简而言之，如果调用了一个修改操作，指向字符串的引用一定会失效，因为你永远不知道底层的内存是否被移动——从调用代码来看，这种移动是不可见的。

基于这一事实，在例 15-2 中，对 `s` 的修改操作无论如何都会使得 `rc` 无效。所以，不要仅仅因为会有人想起指向 `s` 的引用，就将 `s` 标记为“永远不可共享”；相反，我们可以只是将它标记为“下一次修改操作之前不可共享”就可以了，因为在修改操作之后，这些被用户想起的引用无论如何都是无效的。对于用户来说，其行为是相同的。

增加了新成员函数后，有其它什么成员需要修改吗？给出理由。

现在我们可以知道，答案是：有。

首先，我们要能够记住一个给定的字符串当前是不是不能共享；如果不能共享，我们在拷贝它的時候就不会使用引用计数。我们可以增加一个布尔标志，但即使这样的成本也应该避免，所以，基于“`refs == unsigned int 的最大可能值`”意味着“不可共享”这一认识，我们可以将“不可共享”的状态信息直接嵌入到变量 `refs` 中。我们还在 `AboutToModify()` 中增加一个标记，表示是否要将字符串标记为不可共享。

```
// 正确的方案: 对于 operator[] 的第三个尝试
//
// 为了方便，增加了一个新的静态成员，
// 并对 AboutToModify() 做了适当修改。
// 因为我们现在需要在不止一个函数中复制
// StringBuf (见下面 String 的拷贝构造函数)，
// 我们还要将这一逻辑放进一个单独的函数中……
// 不管怎样，StringBuf 现在得有一个自己
// 的拷贝构造函数。
//
const size_t String::Unshareable = numeric_limits<size_t>::max();
```

```

StringBuf::StringBuf( const StringBuf& other, size_t n )
    : buf(0), len(0), used(0), refs(1)
{
    Reserve( max( other.len, n ) );
    copy( other.buf, other.buf+other.used, buf );
    used = other.used;
}

void String::AboutToModify(
    size_t n,
    bool markUnshareable /* = false */
)
{
    if( data_->refs > 1 && data_->refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        --data_->refs;           // 现在，所有实际工作已经完成，
        data_ = newdata;         // 所以，获得拥有权。
    }
    else
    {
        data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
}

char& String::operator[]( size_t n )
{
    AboutToModify( data_->len, true );
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    return data_->buf[n];
}

```

要注意到，其它所有对 `AboutToModify()` 的调用都像最初所写的那样，可以继续使用。

现在要做的是：如果设置了“不可共享”状态值，我们还要让 `String` 的拷贝构造函数来使用它：

```

String::String( const String& other )
{
    // 如果可能，使用 copy-on-write;
    // 否则，立即执行深拷贝。
    //
    if( other.data_->refs != Unshareable )
    {
        data_ = other.data_;
        ++data_->refs;
    }
}

```

```

    else
    {
        data_ = new StringBuf( *other.data_ );
    }
}

```

String 的析构函数也需要做小小的改动:

```

String::~String()
{
    if( data_->refs == Unshareable || --data_->refs < 1 )
    {
        delete data_;
    }
}

```

String 的其它函数还是像最初所写的那样:

```

String::String() : data_(new StringBuf) { }

void String::Append( char c )
{
    AboutToModify( data_->used+1 );
    data_->buf[data_->used++] = c;
}

}

```

大致就是如此……如果仅限于单线程环境的话，在“缓式优化”短系列的最后一个条款中，我们将看到多线程如何影响我们的 copy-on-write 字符串。请阅读后面条款 16 的详细讨论。

总结

下面，我们将所有代码合在一起。请注意，我趁这个机会对 StringBuf::Reserve() 的实现做了一点修改，它现在会对所选择的“新缓冲区大小”进行上舍入 (round up) 计算，使它的值增大至下一个“4 的整数倍”，从而保证内存缓冲区的大小总是 4 字节的倍数。这都是为了效率，因为很多常见的操作系统无论如何也不会分配比这还小的内存块。对于小字符串来说，这个版本的速度也比当初的版本要快一些。（对最初的版本来说，在指数增长策略真正介入之前，它会分配一个 1 字节缓冲区，然后是一个 2 字节缓冲区，3 字节缓冲区，4 字节缓冲区，6 字节缓冲区。下面版本的代码则直接进入 4 字节缓冲区，然后是 8 字节缓冲区，等等。）

```

namespace Optimized {

class StringBuf
{
public:

```

```

StringBuf();
// 开始为空

~StringBuf();
// 删除缓冲区

StringBuf( const StringBuf& other, size_t n = 0 );
// 初始化为 other 的拷贝,
// 并保证 len >= n

void Reserve( size_t n );
// 保证 len >= n

char* buf;
// 分配的缓冲区
size_t len;
// 缓冲区长度
size_t used;
// 实际使用的字符数
unsigned refs;
// 引用计数

private:
// 禁止拷贝
//
StringBuf( const StringBuf& );
StringBuf& operator=( const StringBuf& );
};

class String
{
public:
String();
// 开始为空
~String();
// 递减引用计数
// (若 refs==0, 删除缓冲区)
String( const String& );
// 指向同一缓冲区,
// 并递增引用计数
void Append( char );
// 添加一个字符
size_t Length() const;
// 字符串长度
char& operator[](size_t);
// 元素访问
const char operator[](size_t) const;

// ...省略 operator=(...) 等...

private:
void AboutToModify( size_t n, bool bUnshareable = false );
// 缓式拷贝, 保证 len>=n,
// 若不可共享则做标记
static size_t Unshareable; // "不可共享" 的引用计数标记
StringBuf* data_;
};

StringBuf::StringBuf()
: buf(0), len(0), used(0), refs(1) { }

StringBuf::~StringBuf() { delete[] buf; }

StringBuf::StringBuf( const StringBuf& other, size_t n )
: buf(0), len(0), used(0), refs(1)
{
Reserve( max( other.len, n ) );
copy( other.buf, other.buf+other.used, buf );
used = other.used;
}
}

```

```

void StringBuf::Reserve( size_t n )
{
    if( len < n )
    {
        // 和条款 14 相同的缓冲区增长代码；但在现在的版本中，
        // 新的缓冲区字节数被上舍入至最接近的 4 的倍数。
        size_t needed = max<size_t>( len*1.5, n );
        size_t newlen = needed ? 4 * ((needed-1)/4 + 1) : 0;
        char* newbuf = newlen ? new char[ newlen ] : 0;
        if( buf )
        {
            copy( buf, buf+used, newbuf );
        }

        delete[] buf;    // 现在，所有实际工作已经完成
        buf = newbuf;    // 所以，获得拥有权
        len = newlen;
    }
}

const          size_t           String::Unshareable      =
numeric_limits<size_t>::max();

String::String() : data_(new StringBuf) { }

String::~String()
{
    if( data_->refs == Unshareable || --data_->refs < 1 )
    {
        delete data_;
    }
}

String::String( const String& other )
{
    // 如果可能，使用 copy-on-write;
    // 否则，立即执行深拷贝
    //
    if( other.data_->refs != Unshareable )
    {
        data_ = other.data_;
        ++data_->refs;
    }
    else
    {
        data_ = new StringBuf( *other.data_ );
    }
}

void String::AboutToModify(
    size_t n,
    bool  markUnshareable /* = false */)

```

```

}

{
    if( data_->refs > 1 && data_->refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        --data_->refs;           // 现在，所有实际工作已经完成
        data_ = newdata;          // 所以，获得拥有权
    }
    else
    {
        data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
}

void String::Append( char c )
{
    AboutToModify( data_->used+1 );
    data_->buf[data_->used++] = c;
}

size_t String::Length() const
{
    return data_->used;
}

char& String::operator[]( size_t n )
{
    AboutToModify( data_->len, true );
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    return data_->buf[n];
}
}

```

条款 16：缓式优化，之四：

多线程环境

难度：8

在这个短系列的最后一个条款中，我们来分析线程安全对 copy-on-write 字符串的影响。罗宾所见真的是一种优化吗？答案可能会让你大吃一惊。

(1) Optimized::String（来自条款 15）为什么不是线程安全的？举例说明。

(2) 针对下面两种情况，分别演示如何使 Optimized::String 变得线程安全：

- (a) 假设可以使用“获取（get）”、“设置（set）”和“比较（compare）”整数值的原子操作（atomic operation）；
 (b) 假设没有以上原子操作。
 (3) 在性能上有哪些影响？请讨论。

解答

Copy-on-Write (COW) 的线程安全问题

标准 C++ 对线程这一主题上只字不提。不像 Java，C++ 没有对线程提供内部支持，它没打算通过语言去处理线程安全（thread-safety）问题。那么，为什么要专门准备一个条款来讨论有关线程（thread）和并发处理（concurrency）的议题呢？这只是因为，越来越多的程序员在写多线程（MT）程序，如果在讨论 copy-on-write String 的实现时不涉及线程安全的主题，这样的讨论就是不完整的。几乎从 C++ 本身诞生之日起，就一直有很多可以很好地处理线程的 C++ 类（class）和程序框架（framework）。到了今天，它们已经包括很多功能设施，例如 ACE^⑯、作为 omniORB^⑰一部分的 omnithread 类库，以及很多商用程序库所提供的那些功能设施。你可能一直在使用这其中的某个程序库；或者，你在使用你们自己内部开发的程序库，它包装了你的操作系统的原始服务功能。

在阅读本条款时，还请参阅本书“附录 A：（在多线程环境下）并非优化”。该附录和本条款的分析结果一致，而且建立在本条款的分析结果之上。它不但向你演示错误的优化会对你和你的产品代码带来哪些影响，还分析了如何去发现和避免那些影响。本条款提供了分析结果的概要，更详细的结果呈现在本书“附录 B：单线程 String 实现与多线程安全 String 实现的对比测试结果”。

1. Optimized::String（来自条款 15）为什么不是线程安全的？举例说明。

假设一段代码使用了某个对象，如果在必要的时候，对这个对象的访问需要确保被串行化（serialized），那么，这种串行化的责任要由这段代码来承担。例如，如果某个 String 对象可以被两个不同的线程修改，可怜的 String 对象是没有责任去防止它自己被滥用的。要保证两个线程永远不会在同一时间修改同一个 String 对象，其责任在于使用 String 对象的那段代码。

在条款 15 的代码中，问题表现为两点。首先，之所以提供 copy-on-write (COW) 实现，其目的在于隐藏一个事实，即，两个不同的可见 String 对象可以共享同一个隐藏状态。所以，要想保证一个内部实体（representation）被共享的 String 对象永远不会被

⑯ <http://www.gotw.ca/publications/mxc++/ace.htm>

⑰ <http://www.gotw.ca/publications/mxc++/omniorb.htm>

调用者代码修改，其责任在于 String 类。前面演示的 String 代码已经完全做到了这一点，即，当一个调用者试图修改 String 的时候，如果有必要，它会执行深拷贝（从而使得实体不可共享）。大体来说，这一点没有问题。

不幸的是，这给我们带来第二个问题：在 String 中，使得实体“不可共享”的代码不是线程安全的。想象一下，如果有两个 String 对象 s1 和 s2：

- (a) s1 和 s2 刚好都共享底层的同一个实体（没问题，这正是设计 String 的初衷）；
- (b) 线程 1 试图修改 s1（没问题，因为线程 1 知道，没有其它线程正在修改 s1）；
- (c) 线程 2 试图修改 s2；
- (d) 以上两个操作同时进行（出问题了）。

问题出在(d)。在同一时间，s1 和 s2 都试图使得共享实体“不可共享”，但完成这一操作的代码不是线程安全的。让我们具体看看 String::AboutToModify()中最开头的那行代码：

```
void String::AboutToModify(
    size_t n,
    bool markUnshareable /* = false */
)
{
    if( data_->refs > 1 && data_->refs != Unshareable )
    {
        /* .....其它代码 ..... */
    }
}
```

这个 if 条件语句不是线程安全的。首先，“`data_->refs > 1`”运算甚至都有可能是原子 (atomic) 操作。这样的话，当线程 1 试图对 “`data_->refs > 1`”求值而同时线程 2 正在更新 refs 值的时候，`data_->refs` 读出的值有可能是任何东西——1, 2，甚至既不是原始值，也不是新值。问题在于，String 没有遵循最起码的线程安全要求，即，如果在代码中使用了一个对象，这段代码就必须保证：在需要的时候，对对象的访问必须被串行化。在这里，String 必须保证：不会有两个线程在同一时间以危险的方式使用同一个 refs 值。要做到这一点，常规方法是使用互斥体 (mutex) 或信号量 (semaphore)，借助它们来串行访问被共享的 StringBuf (或者，只是它的 refs 成员)。当前情况下，至少得保证“比较一个 int”的操作是原子操作。

这将我们带到第二个问题面前：即使读取和更新 refs 的操作是原子操作，甚至即使程序是在单 CPU 的机器上运行，从而使得线程是交替执行 (interleaved) 而非真正地并行 (concurrent)，但一个事实还是存在：这个 if 条件语句有两个部分。这样就有一个问题，执行上面代码的线程在计算了条件语句中的第一部分后可能会被中断，但此时第二部分还没有开始计算。在例子中：

线程 1	线程 2
进入 s1 的 AboutToModify() 对 “data_->refs > 1” 求值 (结果为 true, 因为 data_->refs 为 2)	
执行环境切换	
	进入 s2 的 AboutToModify() (运行至结束, 包括将 data_->refs 递减至 1) 退出 s2 的 AboutToModify()
执行环境切换	
对 “data_->refs != Unshareable” 求值 (结 果为 true, 因为 data_->refs 现在为 1) 进入 AboutToModify()的 “我正被共享并 需要取消共享” 代码块, 复制实体, 将 data_->refs 递减至 0, 并放弃掉指向 StringBuf 的最后一个指针。糟糕, 出现了 内存泄漏, 因为曾被 s1 和 s2 共享的 StringBuf 现在根本无法被删除。 退出 s1 的 AboutToModify()	

有了以上分析, 我们来看看如何解决这些安全问题。

保护 COW String

请记住, 在以下所有讨论中, 我们的目标不是想防止“对 Optimized::String 的各种可能的滥用”。毕竟, 要想知道某一可见的 String 对象是否可以被不同的线程使用, 并在需要的时候进行正确的并发控制, 这还是得由使用 String 的代码来负责——这再正常不过。这里的问题在于, 由于 String 的底层增加了额外的共享机制, 调用者代码无法履行其正常的保护职责, 因为它不知道哪个 String 对象真的是完全独立的, 或者哪个对象只是看起来是独立的, 但实际上在底层和其它对象有牵连。所以, Optimized::String 的目标只不过是回到自己的岗位上, 在它自己所在的层次上做到最大的安全; 这样一来, 其它代码在使用 String 时, 就可以安全地认为不同的 String 对象真的是独立的, 从而, 这些代码也就可以根据需要履行其正常的并发控制职责。

2. 针对下面两种情况，分别演示如何使 Optimized::String 变得线程安全：

- (a) 假设具有获取 (get)、设置 (set) 和比较 (compare) 整数值的原子操作 (atomic operation)；
- (b) 假设没有以上原子操作。

我先回答问题(b)，因为它更具普遍性。在这里，我们需要的是一个锁定管理设备，如互斥体。^⑩ 要做的工作非常简单。如果事情一切正常，我们只需锁定“对引用计数本身的访问”。

在开始其它工作之前，需要在 Optimized::StringBuf 中增加一个 Mutex 成员对象。让我们将这个成员称为 m：

```
namespace Optimized
{
    class StringBuf
    {
        public:
            StringBuf();                                // 开始为空
            ~StringBuf();                               // 删除缓冲区
            StringBuf( const StringBuf& other, size_t n = 0 );
                // 初始化为 other 的拷贝,
                // 并保证 len >= n

            void Reserve( size_t n ); // 保证 len >= n

            char*     buf;                      // 分配的缓冲区
            size_t    len;                      // 缓冲区长度
            size_t    used;                     // 实际使用的字符数
            unsigned refs;                    // 引用计数
            Mutex     m;                       // 对本对象执行串行化工作

        private:
            // 禁止拷贝
            //
            StringBuf( const StringBuf& );
            StringBuf& operator=( const StringBuf& );
    };
}
```

必定会同时操作两个 StringBuf 对象的函数只有一个，即拷贝构造函数。String 只会在两个地方调用 StringBuf 的拷贝构造函数 (String 自身的拷贝构造函数中，以及 AboutToModify() 中)。注意，String 只需对引用计数访问进行串行化处理；因为根据定义，没有 String 会对共享的 StringBuf 进行任何操作 (被共享后，如果想得到它的拷贝，

^⑩ 如果你在使用 Win32，所谓的“关键段”（这个术语有点欺骗性）会比互斥体更有效率，因而一般情况下你应该使用关键段，除非你真的需要 Win32 互斥体这一重量级工具。

我们尽可以对其进行读取。但我们不必担心会有人试图改变这个缓冲区，或是对它执行 Reserve() 操作，或是用别的什么方法修改 / 移动它。)

缺省构造函数不需要加锁：

```
String::String() : data_(new StringBuf) {}
```

析构函数只需对 refs 计数值的查询和更新操作加锁：

```
String::~String()
{
    bool bDelete = false;
    data_->m.Lock(); //-----
    if( data_->refs == Unshareable || --data_->refs < 1 )
    {
        bDelete = true;
    }
    data_->m.Unlock(); //-----
    if( bDelete )
    {
        delete data_;
    }
}
```

请注意，对于 String 的拷贝构造函数，我们可以假设，在这个操作期间，其它 String 的数据缓冲区不会被修改或移动；因为，对可见对象的访问进行串行化是调用者的责任。但是，对引用计数本身的访问，我们还是得进行串行化处理，做法和前面一样：

```
String::String( const String& other )
{
    bool bSharedIt = false;
    other.data_->m.Lock(); //-----
    if( other.data_->refs != Unshareable )
    {
        bSharedIt = true;
        data_ = other.data_;
        ++data_->refs;
    }
    other.data_->m.Unlock(); //-----

    if( !bSharedIt )
    {
        data_ = new StringBuf( *other.data_ );
    }
}
```

所以，要想使 String 的拷贝构造函数变得安全，一点都不难。那么，再来看看 AboutToModify()，它也很类似。但要注意，在这里的示例代码中，整个深拷贝操作实际上都被加锁了——确实，严格来说只需要在两个地方加锁，即，查看 refs 值以及最后更新 refs 值的地方。但我们还是对整个操作加锁吧，不要为了获得一点点并发处理

上的好处而在深拷贝期间解锁，然后又仅仅为了更新 refs 而去对它加锁：

```
void String::AboutToModify(
    size_t n,
    bool markUnshareable /* = false */
)
{
    data_->m.Lock(); //-----
    if( data_->refs > 1 && data_->refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        --data_->refs;      // 现在，所有实际操作已经完成，
        data_->m.Unlock(); //-----
        data_ = newdata;     // 所以，获得拥有权
    }
    else
    {
        data_->m.Unlock(); //-----
        data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
}
```

其它函数都不需要修改。Append()和operator[]()不需要加锁，因为一旦AboutToModify()执行结束，我们所使用的就必然不是一个被共享的实体。Length()不需要加锁，因为根据定义，如果我们的StringBuf没被共享，我们会没事儿（没有其他人会修改我们使用的字符串长度值）；如果它被共享，我们也没事儿（另一个线程在对它进行操作之前会先让自己拥有一个拷贝，所以还是不会修改我们使用的计数）：

```
void String::Append( char c )
{
    AboutToModify( data_->used+1 );
    data_->buf[data_->used++] = c;
}

size_t String::Length() const
{
    return data_->used;
}

char& String::operator[]( size_t n )
{
    AboutToModify( data_->len, true );
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    return data_->buf[n];
}
```

再次注意其中有趣的一点：需要加锁的地方只涉及到 refs 计数本身。

有了这一观察结果，并在了解了上面的一般性方案之后，让我们回过头来看看问题的(a)部分，即：

a) 假设可以使用“获取 (get)”、“设置 (set)”和“比较 (compare)”整数值的原子操作 (atomic operation)：

一些操作系统提供了这类函数。

注意：一般来说，这些函数比通用的同步原语如互斥体的效率要高得多。然而，如果说可以用原子整数操作 (atomic integer operation) 来“代替加锁机制”，这也是荒谬的，因为加锁机制还是很有必要——通常来说，这种加锁机制的开销没有其它方式昂贵；但它也不是完全免费的，这一点我们将在后面看到。

这里是一个线程安全的 String 实现，它使用了三个函数：IntAtomicGet()、IntAtomicDecrement() 和 IntAtomicIncrement()，这三个函数都可以安全地返回新值。我们本质上是在做与前面相同的事，但在这里，我们只使用原子整数操作来串行访问 refs 计数。

```
namespace Optimized
{
    String::String() : data_(new StringBuf) { }

    String::~String()
    {
        if( IntAtomicGet( data_->refs ) == Unshareable ||
            IntAtomicDecrement( data_->refs ) < 1 )
        {
            delete data_;
        }
    }

    String::String( const String& other )
    {
        if( IntAtomicGet( other.data_->refs ) != Unshareable )
        {
            data_ = other.data_;
            IntAtomicIncrement( data_->refs );
        }
        else
        {
            data_ = new StringBuf( *other.data_ );
        }
    }
}
```

```

void String::AboutToModify(
    size_t n,
    bool markUnshareable /* = false */
)
{
    int refs = IntAtomicGet( data_->refs );
    if( refs > 1 && refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        if( IntAtomicDecrement( data_->refs ) < 1 )
        {
            // 防止两个线程同时
            delete newdata; // 执行此操作
        }
        else
        {
            // 现在，所有实际工作已经完成
            data_ = newdata; // 所以，获得拥有权
        }
    }
    else
    {
        data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
}

void String::Append( char c )
{
    AboutToModify( data_->used+1 );
    data_->buf[data_->used++] = c;
}

size_t String::Length() const
{
    return data_->used;
}

char& String::operator[]( size_t n )
{
    AboutToModify( data_->len, true );
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    return data_->buf[n];
}
}

```

3. 在性能上有哪些影响？请讨论。

没有原子整数操作，copy-on-write 往往会招致重大的性能损失。即使有了原子整数操作，COW 也会使一个普通的 String 操作的耗时增长将近 50%，甚至在单线程程序中也是如此。

一般来说，如果准备将代码用于多线程环境，在这样的代码中使用 copy-on-write 通常是个坏主意。原因在于，对于两个不同的 String 对象是否在底层实际共享同一实体，调用者代码不再可能知晓；所以，String 必须通过一定的开销完成必要的串行化工作，这样，调用者代码才有可能在线程安全方面承担其正常的责任。至于性能上的影响到底有多大，这取决于是否存在更具效率的实现手段，如原子整数操作；有无这样的实现手段，其性能影响的差异可以从一般到巨大。

一些测试结果

在这个测试环境中，我对以下六种主要形式的字符串实现进行了测试：

名称	说明（前面代码的改进版本）
Plain	未使用计数的字符串；以下所有版本都以此为原型
COW_Unsafe	Plain + COW，非线程安全
COW_AtomicInt	Plain + COW + 线程安全
COW_AtomicInt2	COW_AtomicInt + StringBuffer 和数据在相同的缓冲区
COW_CritSec	Plain + COW + 线程安全（Win32 关键段）
COW_Mutex	Plain + COW + 线程安全（Win32 互斥体）（采用 Win32 互斥体而非 Win32 关键段的 COW_CritSec）

我还提供了第七种形式，以测试“内存分配优化”的结果，而非“拷贝优化”。

Plain_FastAlloc Plain + 优化的内存分配程序

我将注意力集中在 Plain 和 COW_AtomicInt 的对比上。通常来说，在线程安全的 COW 实现中，COW_AtomicInt 最有效率。最终测试结果如下：

(1) 对所有的修改操作 (mutating operations) 和可能要进行修改的操作 (possibly-mutating operations) 来说，COW_AtomicInt 总是差于 Plain。这很自然，而且可以预计得到。

(2) 如果有很多未被修改的拷贝，COW 本应该表现出色。但是，对于一个平均长度为 50 的字符串来说：

(a) 当 33% 的拷贝从未被修改、其余的每个拷贝只被修改一次时，COW_AtomicInt 还是比 Plain 慢。

(b) 当 50% 的拷贝从未被修改、其余的每个拷贝只被修改三次时，COW_AtomicInt 还是比 Plain 慢。

对很多人来说，第二个结果会更令人吃惊——特别是，当整个系统中拷贝操作比修改操作多时，COW_AtomicInt 还是比 Plain 慢！

注意，在上面两种情况下，传统的非线程安全的 COW 确实比 Plain 表现更佳。这说明，对于纯粹的单线程环境来说，COW 可以是一种优化，但它不总是适用于具有线程安全的代码。

(3) 说 COW 的主要优点在于避免内存分配，这是个美丽的谎言。COW 的主要优点在于，它避免了字符串内字符的拷贝，对长字符串来说尤其如此。

(4) 在所有场合下，内存分配优化始终是一种真正的速度优化（但要注意，这是以空间代价作为交换条件的），COW 则并非如此。以下叙述可能是从附录的“详细测试”部分得出的最重要的结论：

COW 在小字符串上的主要优点大多数都可以无需使用 COW 而得到，即，可以通过使用一种更有效率的内存分配程序来得到。（当然，你也可以二者兼施——同时使用 COW 和高效的内存分配程序。）

让我们简要地提出两个很自然的问题并作出回答。

第一个问题，为什么要对 COW_CritSec 这种缺乏效率的方案也进行测试？答案很简单：因为，尽管 COW_CritSec 几乎总是一种“恶化”行径，但直到 2000 年，至少还有一个很普及的商用 basic_string 实现使用了这种方法（也许目前还是这样，我近来没有看过它的代码）。一定要检查一下自己使用的程序库，看看你的供货商是不是也在这么做；因为，如果在建立（build）程序库时考虑到了多线程环境下的使用，那么，即使你的程序是单线程的，你也总得忍受它所招致的性能损失。

第二个问题，什么是 COW_AtomicInt2？COW_AtomicInt2 和 COW_AtomicInt 几乎相同，但它没有在分配 StringBuf 之后去单独分配数据缓冲区，而是使 StringBuf 和数据分配在相同的内存块中。请注意，其它所有的 COW_* 实现都为 StringBuf 对象提供了快速内存分配器（从而避免了不合算的两次分配开销），测试 COW_AtomicInt2 的目的主要是为了说明，我的确考虑到了这一点。对于大多数操作，COW_AtomicInt2 确实比 COW_AtomicInt 慢一点，因为它多了一些额外的处理逻辑。

因为低劣的实现或函数调用的开销可能会给 COW_AtomicInt 的测试结果带来偏差，为了公正，我还对各种整数操作（递增 int、递增 volatile int、使用 Win32 原子整数操作递增 int）的性能进行了测试和对比。

总结

关于测试程序及测试结果的几点重要说明：

- (1) 读者留意：这里的测试程序尚不完整，请将它看作测试工具的雏形。欢迎批评和指正。我在这里展示的是原始的性能测试数据，我没有检查编译后的机器码，也没有测试高速缓存命中率或差错率带来的影响，以及其它负效应。
- (2) 世上没有免费的午餐——Robert A. Heinlein。即使有原子整数操作，也不能说“不需要加锁机制”。因为原子整数操作确实执行了串行化操作，也确实带来了显著的开销。
- (3) 世上没有绝对的公平——Larry Niven。测试程序本身是单线程的。即使在“本身不是多线程的程序”中，线程安全的 COW 实现也会带来开销。充其量，只有在完全不需要让 COW 代码变得线程安全的时候，COW 才会是一种真正的优化（即便如此，请参见[Murray93]第 72-73 页：更多的实验数据显示，COW 只是在某些情况下会带来好处）。如果需要线程安全，COW 会将巨大的性能损失强加给所有的用户——即使用户只是运行单线程程序。

异常安全议题及技术

在现代 C++ 程序设计中，对异常安全（exception safety）议题一无所知却想写出健壮的代码，这无异于痴人说梦。的确如此。

如果你在使用 C++ 标准库，哪怕只是用到了 new，你也得为异常做好准备。本章建立在 Exceptional C++ 相应章节的基础上，并讨论了新的议题及相关技术，例如：std::uncaught_exception() 是什么？它能帮助你写出更健壮的代码吗？异常安全影响到类的设计吗？或者，它可以只是作为事后的改进手段来使用吗？为什么要用管理者对象（manager objects）来封装资源所有权？为什么“资源获取才是初始化”的认识对于编写安全代码如此重要？

但首先还是让我们来熟悉一下，看一个有关异常安全的基本话题：这个话题对以下非常重要的 C++ 基本概念进行了说明，并剖析了其更深层的含义：构造具有什么含义？什么是对象的生命期？

条款 17：构造函数失败，之一：

对象生命期

难度：4

问题：如果由于产生异常而到底会发生什么？如果异常来自于对资源取得对象的析构，情况又会怎样？

(I) 请看下面这个类：

```
// 例 17-1
//
class C : private A
{
    B b_;
};
```

在 C 的构造函数中，如何捕捉从基类子对象（例如 A）或成员对象（例如 b_）的构造函数中抛出的异常？

(2) 请看下面的代码：

```
// 例 17-2
//
{
    Parrot p;
}
```

这个对象的生命期何时开始？何时结束？在对象的生命期之外，对象处于什么状态？最后，如果它的构造函数抛出了一个异常，那将意味着什么？

解答

在 C++ 中增加 function try block 这一特性后，函数可以捕获异常的范围稍微增大了一些。本条款的内容涉及：

- 在 C++ 中，对象构造与构造失败的含义；
- 基类或成员子对象的构造函数抛出异常时，function try block 可用于转化（而非抑制）这个异常。

为方便起见，除非特别指出，本条款中的“成员”指的是“非静态的类数据成员”。

Function Try Block

1. 请看下面这个类：

```
// 例 17-1
//
class C : private A
{
    B b_;
};
```

在 C 的构造函数中，如何捕捉从基类子对象（例如 A）或成员对象（例如 b_）的构造函数中抛出的异常？

这正是 function try block 的用武之地：

```
// 例 17-1(a): 构造函数的 function try block
//
C::C()
```

```

try
: A ( /*...*/ ) // 可选的初始化列表
, b_ ( /*...*/ )
{
}
catch( ... )
{
    // 一旦 A::A() 或 B::B() 抛出异常，我们会来到这儿

    // 如果 A::A() 成功，然后 B::B() 抛出异常，
    // C++语言将保证，在到达本 catch block
    // 之前，A::~A() 会被调用，以摧毁已经创建
    // 的基类 A 子对象。
}

```

然而，更有趣的问题是：你为什么会想到这么做？这个问题引出了本条款两大要点的第一个。

对象生命期与构造函数异常的含义

过一会儿，我们将回答一个问题，即，上面 C 的构造函数是否可以（或应该）吸收 (absorb) A 或 B 的构造函数产生的异常，从而完全不发出异常。在可以正确回答这个问题之前，我们需要完全了解对象生命期^⑩的概念，以及构造函数抛出异常的含义。

2. 请看下面的代码：

```

// 例 17-2
//
{
    Parrot p;
}

```

这个对象的生命期何时开始？何时结束？在对象的生命期之外，对象处于什么状态？最后，如果它的构造函数抛出了一个异常，那将意味着什么？

让我们一次回答一个问题：

问：一个对象的生命期何时开始？

答：当它的构造函数成功执行完毕并正常返回之时。也就是说，当控制 (control) 抵达构造函数体的末尾之时，或执行完一个更早的 return 语句之时。

问：一个对象的生命期何时结束？

^⑩ 为简化起见，我在这里所说的只是类型为 class、具有构造函数的对象的生命期。

答：当它的析构函数开始执行之时，也就是说，当控制抵达析构函数体的开始处之时。

问：对象的生命期结束之后，对象处于什么状态？

答：我们的回答正如一位知名软件大师曾经表述的那样：在谈到一段类似的代码时，他将局部对象（local object）拟人化地称为“他”：

```
// 例 17-3
//
{
    Parrot& perch = Parrot();
}
// <-- 独白从这里开始
```

他并非日渐消瘦！他已经逝去！这只鹦鹉（Parrot）已经芳踪不再！他已经停止了生命！他已经死亡，去见他的造物者去了！他是死尸！被剥夺了生命，安静长眠！如果你没有把他放在枝头（perch），它已经命归黄土！[甚至更早，在这个代码块尾部之前]他的代谢过程已经作古！他离开了枝头！他已经撒手人寰，摆脱了尘世的烦恼，拉上了生命的帷幕，加入了唱师班，无影无踪！这是一只前世的鹦鹉（ex-Parrot）！

——Dr. M.Python, B.Math, MA.Sc., Ph.D. (CompSci) ^②

撇开玩笑不说，此处的重点在于：在生命期开始之前与生命期结束之后，对象的状态完全一样——没有对象存在。就是这样。这一结论将我们带到第二个重要问题前：

问：从构造函数中抛出异常意味着什么？

答：这意味着构造已经失败，对象从没存在过，它的生命期从没开始过。确实，报告构造失败——也就是说，无法正确构造出某种类型的有效对象——的唯一方法是抛出一个异常。（是的，有一条如今已经过时的编程规则是这么说的：“如果程序出了问题，可以将一个状态标志设为‘bad’，让调用者通过一个 IsOK() 函数去检查它”；后面，我会对此谈谈我的看法。）

顺便说一句，如果构造函数不成功，析构函数就永远不会被调用，其原因正在于此——没有东西可以摧毁。它无法死亡，因为它从来就未曾生存过。请注意，这样一来，“一个对象的构造函数抛出异常”这句话实际上具有矛盾性。这样一种东西甚至不能被称为一个前对象（ex-object）。它从没有生存过，从没有加入过对象家族。它是一个非对象（non-object）。

^② 向 Monty Python 致歉。

对于 C++ 的构造函数模型，我们可以总结如下：

只会是二者之一：

(a) 构造函数正常返回，即，控制抵达函数体的尾部，或者执行了一个 `return` 语句。这种情况下，对象真实存在。

(b) 构造函数抛出异常后退出。这种情况下，对象不仅不会继续存在，而且，实际上它根本就从未作为一个对象存在过。

没有其它的可能性。具备了这些知识，我们就可以更好地应对下一条款中的问题了：可否吸收异常？

条款 18：构造函数失败，之二：

吸收异常？

难度：7

本条款就构造函数失败这一主题进行了详尽的分析，说明了 C++ 规则为什么要以那样的方式制定，并阐述了构造函数异常规范的含义。

(1) 在条款 17 的例 17-1 中，如果 A 或 B 的构造函数抛出异常，C 的构造函数有可能吸收这个异常并完全不发出异常吗？证明你的答案，举例说明为什么那是它应该具有的运作方式。

(2) 为了能够在 C 的构造函数上安全地放上一个空 `throw` 规范，A 和 B 必须满足的最小条件是什么？

解答

我无法挽留未捕获的异常^①

1. 在条款 17 的例 17-1 中，如果 A 或 B 的构造函数抛出异常，C 的构造函数有可能吸收这个异常并完全不发出异常吗？

如果不考虑对象生命期规则，我们可能已经写出下面这样的代码：

^① 双关语，可以加到 Rolling Stone 乐队的“Satisfaction”的合唱部分，或 Pink Floyd 的“Another Brick in the Wall, Part N”的开头几个小节。^②（译注：在这两首歌曲中，作者提到的那两个部分都反复吟唱一句歌词。）

```

// 例 18-1(a): 吸收异常?
//
C::C()
try
    : A ( /*...*/ ) // 可选的初始化列表
    , b_( /*...*/ )
{
}
catch( ... )
{
    // ?
}

```

这个 try block 处理程序 (handler) 将如何退出? 请注意以下几点:

- 这个处理程序不能简单地以 “return;” 返回, 因为这不合法.
- 如果在处理程序中写上 “throw;”, 那么, 无论 A::A() 或 B::B() 最初抛出什么异常, 它都会将其重新抛出.
- 如果处理程序抛出其它某个异常, 最终抛出的也就是那个异常, 而不是基类或成员子对象的构造函数最初抛出的异常.
- 还有一点不太明显, 但在 C++ 标准中表述得很清楚, 这就是: 如果处理程序没有以抛出异常的方式退出 (既没有重新抛出最初的异常, 也没有抛出什么新的东西), 那么, 在控制抵达构造函数或析构函数的 catch block 的末尾时, 最初的异常会被自动地重新抛出, 就像处理程序的最后一个语句是 “throw;” 一样.

想一想这意味着什么: 一个构造函数或析构函数的 function try block 的处理程序代码 “必须” 以抛出某个异常结束. 没有其它方式! 只要不违反异常规范, C++ 语言不关心你抛出的是什么异常——可以是最初的那个, 或者是其它某个经过转化 (translated) 的异常——但必须有个异常! 只要基类或成员子对象的构造函数抛出任何异常, 就一定会导致某个异常从它们的外层构造函数 (containing constructor) 中泄露出来, 想阻止这一点是不可能的.

简而言之, 这意味着:

在 C++ 中, 只要任何一个基类或成员子对象构造失败, 整个对象的构造必然失败.

基类或成员子对象的构造函数抛出异常后, 构造函数不可能恢复正常并合理运作. 它甚至不能将自己的对象置于 “构造失败” 状态, 以让编译器能够识别. 它的对象 “没有” 被构造出来; 它永远也不会被构造出来, 无论处理程序尝试何种 Frankensteinian (毁灭自身) 式的方式想为这个非对象 (nonobject) 注入生命. 任何可以摧毁的东西都已经由语言本身自动摧毁了, 这包括所有的基类和成员子对象.

如果一个类确实可以具有某种合理的 “部分” 构造失败 (construction partially failed) 状态呢? ——也就是说, 它真的具有某些不是绝对需要的 “可选” 成员, 对象可

以没有它们而苟延残喘，只是有可能缺少某些功能而已。这种情况下就可以运用 Pimpl 手法（Exceptional C++ [Sutter00] 条款 27-30 中有详细介绍），在一定的安全距离内拥有对象中可能损坏的部分。与此类似的论述请参见 Exceptional C++ 的条款 31-34 中有关“滥用继承”的讨论。顺便说一句，“对象的可选部分”这一观念也是“无论何时都要尽量使用委托（delegation）而不使用继承（inheritance）”的另一个重要原因。基类子对象永远不能成为可选部分，因为你不可能将基类子对象放进一个 Pimpl^②。

过去，我时常对异常又爱又恨。即便是那样，我也不得不一向承认：鉴于构造函数无法通过返回值报错（大多数运算符都是如此）这一事实，异常是通知构造函数失败的合适方法。对于“如果构造函数出错，就设置一个状态位，让用户调用 IsOK()，从而检查出构造是否真的成功”这一方法，我认为它已经过时，而且危险、乏味，绝对不比抛出异常的方法好。这一方法的另一变形是“两阶段构造法”，其情况也是如此。并不是只有我认为这些方法可恶。对这些过时的编程风格进行最彻底抨击的人正是 Stroustrup，他就此提出了很多辛辣的讽刺，如“伪 C++”、“这种风格是异常(exception)出现之前 C++ 时代的遗迹”等等，具体内容请参见 [stroustrup00] 第 E.3.5 章节。

向法则靠拢

补充一句，这也意味着，构造函数的 function try block 唯一（重复一遍：唯一）可能的用处是“转化（translate）从基类或成员子对象抛出的异常”。这是法则 1。法则 2 则告诉你，析构函数的 function try block 完全没有用处——

“——请等一下！”我听见有人打断了我的话，声音来自房间的中央。“我不同意法则 1。对于构造函数的 function try block，我可以想到另一种可能的应用——有了它，我们就可以释放初始化列表或构造函数体内分配的资源！”

很遗憾，这种说法不对。请记住，一旦你进入构造函数 try block 的处理程序中，构造函数体内的任何局部变量也随之离开生存空间，因而一定不再有基类子对象或成员对象存在。就是这样。你甚至不能引用它们的名称。无论是对象中从未创建的部分，还是已经创建但又被摧毁的部分。所以，对那些需要引用（referring to）到类的基类或成员的东西，你不可能去清除（况且，不管怎么说，这是基类或成员的析构函数要做的事，对吗？）。

^② 收敛性（convergence）的话题有时很有趣。自开始推广 Pimpl 手法并抨击不必要的继承以来，甚至多年之后，我不断碰到新问题，这些问题都通过 Pimpl 手法或舍弃不必要的继承而得到了解决；在改善异常安全问题的时候，尤其如此。我想这没什么奇怪的，因为这只不过是重弹了耦合（coupling）原理的老调而已。更高的耦合性意味着相关部件更容易出错。针对这一看法，Bobby Schmidt 在私人信件中是这样回答的：“也许，这就是我们从这个问题中吸取的主要教训——实际上，我们只不过是重新发现并强调了低耦合高内聚这条古老的公理而已。”

附笔：C++为什么要那样做？

为了理解 C++那样做的好处，让我们暂时将限制抛在一边，假设 C++真的让你可以在构造函数的 try block 处理程序中使用成员的名称。然后再假设下面这种情况，试着作出决定：处理程序应该删除 `t_` 或 `z_` 吗？（再一次，暂时忽略一件事实：在真正的 C++ 中，这个处理程序甚至不能引用 `t_` 或 `z_`。）

```
// 例 18-1(b)：一个错误百出的类
//
class X : Y
{
    T* t_;
    Z* z_;
public:
    X()
    try
        : Y(1)
        , t_( new T( static_cast<Y*>(this) ) )
        , z_( new Z( static_cast<Y*>(this), t_ ) )
    {
        /*...*/
    }
    catch(...) // Y::Y 或 T::T 或 Z::Z
               // 或 X::X 体内抛出异常
    {
        // 提问：应该删除 t_ 或 z_ 吗？
        // (注意：这不是合法的 C++ 代码！)
    }
};
```

第一个问题是，我们不可能知道是否已经分配过 `t_` 或 `z_`。因而删除它们中的任何一个都是不安全的。

第二个问题是，即使知道已经执行了其中一个分配，我们还是不能摧毁 `*t_` 或 `*z_`，因为它们引用了一个不再存在的 Y（也可能是 T），并且还可能去使用那个 Y（也可能是 T）。补充一句，这意味着，不仅是我们不能摧毁 `*t_` 或 `*z_`，实际上它们永远不可能被任何人摧毁。

如果这还没有让你清醒，你也的确应该清醒了。我见过一些人写出的代码，那些代码本质上与上面类似，但这些人从来没有意识到自己正在创建一些永远无法摧毁的对象！这会引发错误的发生。幸运地，有个简单的办法可以避免这个问题。如果 `T*` 成员改由 `auto_ptr` 或类似的管理者对象来保存，这些难题将在很大程度上得以消除。（关于 `auto_ptr` 成员的危险性以及如何避免这些危险，请阅读本书中的条款 30 和 31。）

最后一点，如果 `Y::~Y()` 可以抛出异常，那么，任何时候都不可能可靠地创建 `X` 对象。如果前面你未曾清醒，这一次你一定会如饮醍醐。如果 `Y::~Y()` 会抛出异常，即使写下 “`X x;`” 也会让你危机四伏。这再次强化了那句格言：任何情况下，绝对不允许析构函数产生异常，写一个可以产生异常的析构函数是个不折不扣的错误。析构和异常水火不容。

好了，这一点已经说得够多了。前面的附笔是想让我们有一个更正确的认识，即，C++规则为什么要那样制定。在真正的 C++语言中，你不可能在前面代码中的处理函数内部引用 `t_` 或 `z_`。到目前为止我一直在避免引用标准，所以现在给你一条——来自 C++ 标准[C++98]第 15.3 款第 10 段“在一个对象的构造函数或析构函数的 function try block 处理程序中，引用对象的任何非静态成员或基类将导致不可预测的行为。”

有关 Function Try Block 的法则

所以，至此可以总结如下：

法则 1：构造函数的 function try block 处理程序只能用于转化（translate）从基类或成员子对象的构造函数抛出的异常（也可能做一些相应的记录工作，或其它某种附带性的工作，以响应构造失败）。此外没有其它用途。

法则 2：析构函数的 function try block 鲜有或没有实际用处，因为析构函数绝不应该产生异常^②。所以，绝对不应该有这样一种东西，析构函数的 function try block 能够发现到它，而普通的 try block 却发现不到。即使由于低劣的代码（即，成员子对象的析构函数可以抛出异常）造成有什么东西可以发现得到，处理程序也无法处理这种情况，因为它无法抑制这个异常。它能做的只不过是做些记录，或者发出提示信息。

法则 3：其它所有 function try block 都没有实际用处。对一个函数来说，其内部正常的 try block 不能捕捉到的东西，其正常的 function try block 也不能捕捉得到。

安全地编写代码的法则

法则 4：“获取未管理资源（unmanaged resource）”的操作总是应该放在构造函数体内，绝不要放在初始化列表中。换句话说，要么运用“获得资源才是初始化”的策略（从而完全避免未管理资源的存在），要么在构造函数体内执行资源获取的操作。^③

还是以例 18-1(b)来做说明。假设 `T` 为 `char`，`t_` 是一个普通的旧式 `char*`，它在初始化列表中通过 `new[]` 操作获得资源。这样一来，在异常处理程序中，或是其它任何地方，我们都无法对它执行 `delete[]` 操作。解决之道是，要么将动态分配的内存资源包装起来（例如，将 `char*` 改为 `string`）；要么在构造函数体内执行 `new[]`，这样，在构造函数体内，通过使用局部 try block 或其它方式，它可以被安全地清除。

^② 即使是记录工作或其它附带性工作也不应该，因为基类或成员子对象的析构函数不应该有任何异常。因此，能在析构函数的 function try block 中捕获到的任何东西，同样也可以在析构函数体内的普通 try block 中捕获得到。

^③ 参见[Stroustrup94]第 16.5 节和[Stroustrup00]第 14.4 节。

法则 5：清除“获取未管理资源的操作”总要放在构造函数或析构函数体内的局部 try block 处理程序中，绝对不要放在构造函数或析构函数的 function try block 处理程序中。

法则 6：如果构造函数有异常规范，那么，对于基类和成员子对象可能抛出的所有异常，这个异常规范必须留有余地。正如福尔摩斯会补充的那样：“真的必须这样，你知道”。（的确，隐式生成的构造函数就是以这样的方式声明的；参见[GotW]#69。^②）

法则 7：使用 Pimpl 手法保存类（class）内部的“可选部分”。如果一个成员对象的构造函数可以抛出异常，但你不需要这个成员也可以照常运作，那么，和往常一样，你可以将这个成员用指针来保存，并通过指针是否为 null 来判断你是否得到这个对象。如果运用 Pimpl 手法将这些“可选”成员集中起来，你就只用做一次分配工作。

最后一条法则的内容和其它几条有所重叠，但值得单独列出来重申一遍：

法则 8：尽量使用“获得资源才是初始化”的技术来管理资源。是的！是的！是的！它会消除你很多的头疼病，甚至比你想象的还要多——包括一些难以发现的问题，就像我们在前面讨论过的某些问题一样。

这只不过是它应有的方式

前面讨论了合法性，现在转向合理性：

证明你的答案，举例说明为什么那是它应该具有的运作方式。

一旦考虑到 C++ 的对象生命期模型及其原理，你就会理解：这一语言的运作方式是完全合理的，它可以很容易地“自圆其说”。

构造函数的异常必须被传播。没有其它方式可以表示构造函数失败。一瞬间，两种情形映入脑海：

```
// 例 18-1(c): auto 对象
//
{
    X x;
    g( x ); // 执行其它操作
}
```

如果 X 的构造过程失败——无论是因为 X 自己的构造函数体代码造成的，还是因为 X 的某个基类子对象或成员对象构造失败造成的——控制必然不会在这个代码块范围内继续。毕竟，没有 x 对象存在！控制不在此继续的唯一方式是抛出异常。因此，auto 对象构造失败必然导致抛出某种异常——不管是“造成基类或成员子对象构造失败”

^② 在线提供于：<http://www.gotw.ca/gotw/069.htm>。

的同一个异常，还是某个从 X 构造函数的 function try block 发出的经过转化了的异常。

类似地：

```
// 例 18-1(d)：对象数组
//
{
    X ax[10];
    // ...
}
```

假设第 5 个 X 对象构造失败——无论是因为 X 自己的构造函数体代码失败，或是因为 X 的某个基类或成员子对象构造失败。那么，控制必然不会在这个代码块范围内继续。毕竟，如果想继续，你最终得到的是一个“有洞的数组（holey array）”——即，并非所有对象都真正存在的数组。（即使这可能会比扔下一团糟（holy disarray）不管要好——但我有点扯远了。）

尾声：容错的构造函数？

2. 为了能够在 C 的构造函数上安全地放上一个空 throw 规范，A 和 B 必须满足的最小条件是什么？

请考虑：如果一个类（例如条款 17 例 17-1 中的 C）的某个基类或成员构造函数会抛出异常，有可能为这个类的构造函数写一个空 throw 规范并实施这一规范吗？答案是：不可能；你可以写出“我不会抛出任何东西”这样的空 throw 规范，但这会是个谎言，因为你没有办法去实施它。为了对某一函数实施“不抛出任何异常”的保证，我们就得吸收来自下级（lower-level）代码的所有可能的异常，从而避免意外地将它们发送给我们自己的调用者。如果你真的想写一个“承诺不会抛出异常的构造函数”，你可以绕过“可能会抛出异常的成员子对象”（例如，如果它们确实是可选部分，你可以通过指针或 Pimpl 来保存它们），但你无法躲避“可能会抛出异常的基类子对象”——这也是为什么要避免不必要的继承的另一个原因，因为不必要的继承总是意味着不必要的耦合。

如果想给一个构造函数加上空 throw 规范，我们就得确认：所有的基类和成员子对象绝对不会抛出异常——无论它们有没有提供 throw 规范来表明这一点。构造函数上的空 throw 规范意味着向世界宣告：构造永远不会失败。如果出于某种原因它实际上会失败，那么，这个空 throw 规范就不合适。

如果你为构造函数写了一个空 throw 规范，但某个基类或成员子对象的构造函数真的会抛出异常，那将会发生些什么？简短的回答是：“会去执行 terminate()，会直接执行 terminate()，不会经过 try，不会收到 200 块钱！”稍微详细一点的回答是：函数 unexpected()会被调用，它有两个选择——一个选择是：抛出或重新抛出异常规范所允

许的一个异常（这不可能，因为它是空的，不允许抛出任何异常）；另一个选择是：调用 `terminate()`，而 `terminate()` 会立即中止程序^⑩。用驾驶员的行话来说就是：先是刺耳的急刹车声，然后是嘎吱嘎吱的碎裂声。

总结

在 C++ 中，只是在构造函数成功执行结束后，对象的生命期才开始。因而，从构造函数中抛出异常总是意味着构造失败（这也是唯一一种报告构造失败的方法）。想从基类或成员子对象的构造失败中恢复是不可能的，所以，如果任何基类或成员子对象的构造失败，整个对象的构造必然失败。

避免使用 function try block，这不是因为它们有害，而是因为和普通的 try block 相比，它们提供的好处太少，甚至根本就没有——在你面试应聘者的时候，你会发现，更多的人了解的是普通的 try block，而不是 function try block。这一点遵循了“挑选最有效、最简单的方案”的原则，以及“编写代码时首先考虑清晰性”的原则。只是在转化“从基类或成员的构造函数产生的异常”时，构造函数的 function try block 才有用处。其它所有 function try block 根本就没有什么用处——如果有的话，那实属罕见。

最后，正如 Exceptional C++ 条款 8 至 19 反复强调的那样：使用“有拥有权的对象（owning object）”，并运用“资源获取才是初始化”的策略来管理资源。这样，在你的代码中，你往往可以完全避免写 try 和 catch，更不用说 function try block 了。

条款 19：未捕获的异常

难度：6

`std::uncaught_exception()` 有什么作用？何时该使用它？这里给出的答案会出乎你的意料之外。

(1) `std::uncaught_exception()` 完成什么功能？

(2) 请看下面的代码：

```
T::~T()
{
    if( !std::uncaught_exception() )
    {
        // .....会抛出异常的代码 .....
    }
    else
    {
```

^⑩ 你可以使用 `set_unexpected()` 和 `set_terminate()` 来让你自己的处理程序得以调用，这给了你多做一些记录或清理工作的机会，但它们最终还是得做相同的事。

```
// ……不会抛出异常的代码 ……  
}  
}
```

这是个好办法吗？无论你认为是好还是坏，给出论据。

(3) `uncaught_exception()`有其它什么好的用处吗？讨论并给出结论。

解答

`uncaught_exception()`概述

1. `std::uncaught_exception()`完成什么功能？

标准 `uncaught_exception()` 函数提供了一种方法，让你知道“当前是否有一个异常正处于活动状态”。要注意的重要一点是，这和知道“当前是否可以安全地抛出一个异常”是两码事。

直接引用 C++ 标准中的原文（15.5.3/1）：

从“被抛出对象求值 (*evaluation*) 结束”之时开始，直至在对应的处理程序 (*handler*) 中“异常声明的初始化结束”之时为止，函数 `bool uncaught_exception()` 返回 `true`。这包括堆栈展开 (*stack unwinding*) 时期。如果异常被重新抛出，从被重新抛出的时刻开始，直到重抛的异常被再次捕获为止，`uncaught_exception()` 返回 `true`。

可以看到，这段说明似乎没有什么用处。

背景：抛出异常的析构函数所带来的问题

如果析构函数抛出异常，情况就糟糕了。具体看下面这样一段代码：

```
// 例 19-1：问题  
//  
class X {  
public:  
    ~X() { throw 1; }  
};  
  
void f() {  
    X x;  
    throw 2;  
} // 调用 x::~X (它抛出了异常)，然后调用 terminate()
```

假设一个异常已经处于活动状态（确切地说，处于堆栈展开期间），如果此时析构函数抛出异常，程序会被中止。这通常不是一件好事。

欲了解更多知识，请参阅 *Exceptional C++* [Sutter00] 条款 16，此条款就“会抛出异常的析构函数及其危害原因”进行了讨论。

错误的方案

“啊哈，”很多人，包括很多专家，会说，“为什么不用 `uncaught_exception()` 呢？借助它，我们就可以判断出能否抛出异常！”问题 2 中的代码正源于这一想法。它试图解决前面列出的问题。

2. 请看下面的代码：

```
// 例 19-2：错误的方案
//
T::~T()
{
    if( !std::uncaught_exception() )
    {
        // .....会抛出异常的代码 .....
    }
    else
    {
        // .....不会抛出异常的代码 .....
    }
}
```

这是个好办法吗？无论你认为是好还是坏，给出论据。

简单地回答：不，这不是个好办法，即使它是在试图解决问题。单纯从技术面来看，这个办法就不应该使用。但我更喜欢从行为规范的角度来反驳这一做法。

在例 19-2 的做法中，其背后的思想很简单：只要当前可以安全地抛出异常，我们就选择那条可以抛出异常的执行路径。这一思想错在两个方面。首先，这段代码不会那样做。第二（从我的观点来看，这一点更重要），这一思想本身是错误的。让我们分别探讨这两个方面。

这个错误的方案为什么不正确

第一个问题是，在某些情况下，例 19-2 的做法不会真的像你预想的那样工作。这是因为，即使是在可以安全地抛出异常的时候，它也会采用那条不抛出异常的执行路径。

```
// 例 19-2(a): 这个错误的方案为什么是错的
//
U::~U()
{
    try
    {
        T t;
        // 工作
    }
    catch( ... )
    {
        // 清除
    }
}
```

如果在异常传播期间，由于堆栈展开使得 U 对象被摧毁，T::~T() 将不会使用那条“会抛出异常”的执行路径，即使此时它可以安全地抛出异常。因为 T::~T() 不知道：在这种情况下，它已经受到了外部 U::~U() 中的 catch(...) 代码块的保护。

注意，从本质上来说，例 19-2 和下面的做法没有什么不同：

```
// 例 19-3: 另一个不同形式的错误方案
//
Transaction::~Transaction()
{
    if( uncaught_exception() )
    {
        RollBack();
    }
    else
    {
        // ...
    }
}
```

同样，在堆栈展开期间，如果一个被调用的析构函数用到了 Transaction，这段代码不会做正确的事，如下所示：

```
// 例 19-3(a): 为什么这个不同
// 形式的错误方案还是错的
//
U::~U() {
    try {
        Transaction t( /*...*/ );
        // 工作
    } catch( ... ) {
        // 清除
    }
}
```

所以，例 19-2 不会像你预计的那样工作。言之有理！但这不是主题所在。

这个错误的方案为什么不符合行为规范

这个方案存在的第二个问题更基本，它不是出于技术上的原因，而在于不符合行为规范。让 `T::~T()` 的报错语义具有两种不同“模式”的操作，这不能不说是个糟糕的设计。因为，允许一个操作以两种不同的方式去报告同一个错误，这绝对是个拙劣的设计。使接口不只具有一种形态，而让它处于调用者代码无法轻易控制或理解的形态，这将带来两个重大缺陷。首先，它使得接口和语义复杂化。其次，它使得调用者处境艰难，因为调用者必须能够处理两种不同的报错形式——在太多的调用者本来就不能很好地完成出错检查的时候，情况更是如此。

有时，当我在开车的时候，我会发现自己跟在另一辆车之后，那辆车一半在一条车道上，另一半在另一条车道上。过了一会儿，如果发现这辆车（以我的观点来看）还是处于违规状态，我会忍不住放下车窗，朝外大声、但以一种友好、亲切、关心的语气喊到：“喂，老兄！快挑一条车道！任何一条！”只要那辆车还是继续跨在两条车道上，我就得准备好了它随时转到其中一条车道的可能。这不仅让人觉得讨厌，还限制了我的车速。

有时，当我们在写代码的时候，我们会发现自己在使用另一个程序员提供的类或函数，而这些类或函数却具有精神分裂症般的接口：例如，在报告同一种错误时，它很不明确地采用了多种方式，而不是挑选一种语义然后始终坚持这一语义。于是，我们都会忍不住来到那个罪魁祸首的办公室或工作间，帮助那个可怜的不明就理的程序员做个决定；这样，我们和其他用户才算得以解脱。

正确的方案

例 19-1 中那个问题的正确答案非常简单：

```
// 例 19-4: 正确的方案
//
T::~T() /* throw() */ {
    // .....不会抛出异常的代码 .....
}
```

例 19-4 演示了如何在设计中明作决断，而不作无意义的拖沓。

注意，为了表示不抛出异常，这个 `throw()` 的规范只是作为注释出现在代码中。这是我所采用的编码风格，其部分原因在于：实际上，异常规范没有像它所吹嘘的那样带来多少好处。是否真的要写一个异常规范，这只是你的喜好问题，重要的在于，这个函数真的不会产生异常。关于 `T` 成员的析构函数抛出异常的可能性，请参见前面条款 18 的讨论。

如果需要，T 可以提供一个“用在析构之前的函数”（例如 T::Close()），这个函数可以抛出异常，并针对 T 对象及其拥有的所有资源执行终止（shutdown）操作。采用这种方法，在调用者代码需要检测严重错误的时候，它就可以调用 T::Close(); T::~T() 可以用 T::Close() 加上一个 try/catch block 来实现：

```
// 例 19-5: 另一个正确的方案
//
T::Close()
{
    // ... 会抛出异常的代码 ...
}

T::~T() /* throw() */
{
    try
    {
        Close();
    }
    catch( ... ) { }
}
```

这很好地遵循了“一个函数，一种职责”的原则。最初代码存在的一个问题，是它让同一个函数既负责销毁对象，又要负责最后的清理及报错工作。

还请阅读前面条款 17 和 18，看看为什么我们使用的是析构函数体中的 try block，而不应该是析构函数的 function try block。

设计准则

决不允许异常从析构函数抛出。写析构函数的时候，就像它已经有了一个 throw() 异常规范一样（至于这个 throw 规范是否真的出现在代码中，这纯属个人喜好问题。）

设计准则

如果析构函数调用了一个可能会抛出异常的函数，一定要将这个调用包装在 try/catch block 中，以防止异常逃出析构函数。

3. uncaught_exception() 有其它什么好的用处吗？讨论并给出结论。

抱歉，我不知道 uncaught_exception() 有什么既好又安全的用途。我的建议是：不要用它。

条款 20：未管理指针存在的问题，

之一：参数求值

难度：6

Exceptional C++ 以及本书的读者都知道，异常安全绝非无关紧要。本条款指出了一个直到最近才被发现的异常安全问题，并演示如何最有效地在代码中避免它。

- (1) 在下面每条语句中，你能说出它们对函数 f、g、h 和表达式 expr1、expr2 的求值顺序吗？假设表达式 expr1 和 expr2 不包含进一步的函数调用。

```
// 例 20-1(a)
//
f( expr1, expr2 );
```



```
// 例 20-1(b)
//
f( g( expr1 ), h( expr2 ) );
```

- (2) 当你正在公司的代码库中畅游时，你在一个尘封已久的角度发现了下面这样的代码片段：

```
// 例 20-2
//

// 在某个头文件中：
void f( T1*, T2* );
```



```
// 在某个实现文件中：
f( new T1, new T2 );
```

这段代码有异常安全隐患吗？请解释。

解答

回顾：求值中的有序与无序

1. 在下面每条语句中，你能说出它们对函数 f、g、h 和表达式 expr1、expr2 的求值顺序吗？假设表达式 expr1 和 expr2 不包含进一步的函数调用。

C++标准没有提及线程；所以，不考虑线程的话，第一个问题的答案依赖以下基本规则：

- (1) 在函数调用之前，对函数所有参数的求值必须全部完成。这包括，如果函数参数是表达式，那么，表达式所产生的任何副作用也得全部完成。
- (2) 一旦一个函数开始执行，调用者函数中的表达式将不会开始求值或继续求值，直至被调用函数执行结束。函数执行永远不会交叉（interleave）进行。
- (3) 如果函数参数是表达式，这些表达式通常可以按任何次序求值，包括交叉求值，除非另有其它规则限制。

有了这些规则，我们来看看最开始的例子中会发生些什么：

```
// 例 20-1(a)
// 
f( expr1, expr2 );
```

对于例 20-1(a)，我们能够作出的结论只能是：expr1 和 expr2 都必须在 f() 被调用之前求值。

就这些！编译器可能先对 expr1 求值，也可能后对 expr1 求值，或者让 expr1 和 expr2 的求值交叉进行。很多人觉得这一点很奇怪，新闻组上经常有人就此提出疑问。但这只不过是 C/C++ 中顺序点（sequence points）规则的直接结果。

```
// 例 20-1(b)
// 
f( g( expr1 ), h( expr2 ) );
```

在例 20-1(b) 中，函数和表达式的运算是会遵循以下规则的基础上以任意次序进行：

- expr1 必须在 g() 被调用前求值。
- expr2 必须在 h() 被调用前求值。
- g() 和 h() 都必须在 f() 被调用前执行完毕。
- expr1 和 expr2 的求值可以交叉进行，但任何函数调用不能交叉执行。例如，从 g() 开始执行到结束，不可能有“expr2 中的某部分被求值”或“h() 的某部分在执行”的情况发生。

就是这样！例如，下面这些情况是可能的，甚至还有更多的可能：

- g() 或 h() 都有可能被先调用。
- expr1 开始运算，然后被 h() 的调用中断，然后又继续执行运算，直至结束。（expr2 和 g() 也类似。）

函数调用中的异常安全问题

2. 当你正在公司的代码库中畅游时，你在一个尘封已久的角度发现了下面这样的代码片段：

```
// 例 20-2
//
// 在某个头文件中:
void f( T1*, T2* );
// 在某个实现文件中:
f( new T1, new T2 );
```

这段代码有异常安全隐患吗？请解释。

是的，有好几个异常安全隐患。

简要回顾一下 `new T1` 这种表达式被调用时的情况。这是一个够简单的表达式了，即，`new` 表达式。回忆一下 `new` 表达式实际上做些什么（为简化起见，不考虑 `in-place` 和数组形式，因为它们和现在的话题没有多大关系）：

- 分配内存；
- 在内存中构造一个新的对象；
- 如果由于异常导致构造失败，已分配内存被释放。

所以，每一个 `new` 表达式实质上由两个连续的函数调用构成：首先调用 `operator new()`（要么是全局的那个，要么由被创建对象的类型提供），然后调用构造函数。

对于例 20-2，考虑这样一个问题，如果编译器生成的代码像下面这样，那将会发生什么事：

- (1) 为 `T1` 分配内存；
- (2) 构造 `T1`；
- (3) 为 `T2` 分配内存；
- (4) 构造 `T2`；
- (5) 调用 `f()`。

问题在于：如果由于异常导致第(3)步或第(4)步失败，C++ 标准不会要求 `T1` 对象被摧毁并释放内存。这是典型的内存泄漏，显然不是件好事。

另一种可能的事件顺序是：

- (1) 为 `T1` 分配内存；
- (2) 为 `T2` 分配内存；
- (3) 构造 `T1`；
- (4) 构造 `T2`；
- (5) 调用 `f()`。

这一顺序不只产生一个问题，而是产生两个具有不同后果的异常安全问题：

- (a) 如果因为异常导致第(3)步失败，那么，为 T1 对象分配的内存会被自动释放（第(1)步操作被撤消），但 C++ 标准没有要求释放“为 T2 对象分配的内存”，内存泄漏了。
- (b) 如果因为异常导致第(4)步失败，那么，T1 对象此时已被分配并完全构造，但 C++ 标准没有要求将其摧毁并释放其内存。T1 对象泄漏了。

“嗯？”你会奇怪，“那为什么还允许这种异常安全漏洞存在呢？为什么 C++ 标准不要求编译器在做清除工作时做正确的事，从而防止这一问题呢？”

为了延续 C 在效率上的一贯宗旨，在表达式求值顺序方面，C++ 标准给了编译器一定范围的自由；因为只有这样，编译器才有可能执行优化，否则就不行。因而，对于表达式求值，C++ 标准没有要求做到异常安全；所以，如果想编写具有异常安全的代码，你就得了解这些情况，避免这些情况。幸运的是，你确实可以做到这一点并防止这一问题。或许，auto_ptr 这样的被管理指针（managed pointer）可以给你帮助？我们将在下面的条款 21 中找到答案。

条款 21：未管理指针存在的问题，

之二：使用 auto_ptr?

难度：8

使用 auto_ptr 可以解决条款 20 中的问题吗？

(1) 当你继续在公司的代码库中翻箱倒柜时，你发现，一定有什么人不喜欢条款 20 中的例 20-2，因为那些源文件的后续版本被修改成下面这样：

```
// 例 21-1
//

// 在某个头文件中：
void f( auto_ptr<T1>, auto_ptr<T2> );

// 在某个实现文件中：
f( auto_ptr<T1>( new T1 ), auto_ptr<T2>( new T2 ) );
```

如果这个版本较之条款 20 中的例 20-2 有所改进的话，改进在哪儿？异常安全问题还存在吗？请解释。

(2) 演示如何写一个 auto_ptr_new 工具，用来解决问题(1)中的安全问题，并且可以通过如下方式调用：

```
// 例 21-2
//

// 在某个头文件中:
void f( auto_ptr<T1>, auto_ptr<T2> );

// 在某个实现文件中:
f( auto_ptr_new<T1>(), auto_ptr_new<T2>() );
```

解答

1. 当你继续在公司的代码库中翻箱倒柜时，你发现，一定有什么人不喜欢条款 20 中的例 20-2，因为那些源文件的后续版本被修改成下面这样：

```
// 例 21-1
//

// 在某个头文件中:
void f( auto_ptr<T1>, auto_ptr<T2> );

// 在某个实现文件中:
f( auto_ptr<T1>( new T1 ), auto_ptr<T2>( new T2 ) );
```

如果这个版本较之条款 20 中的例 20-2 有所改进的话，改进在哪儿？异常安全问题还存在吗？请解释。

这段代码试图“抛出^⑩ auto_ptr 来解决问题”。很多人认为，auto_ptr 这样的智能指针是解决异常安全问题的灵丹妙药，或者是试金石或护身符，只要它一现身，编译器的消化不良就会烟消云散。

不是这样。什么也没有改变。例 21-1 依然缺乏异常安全，理由和前面一样。

具体来说，问题在于：只是真正被交给了有管理能力的 auto_ptr 之后，资源才是安全的；但在这里，在任何一个 auto_ptr 的构造函数执行之前，前面说过的相同问题还是照样会发生。因为，前面提到的两个有问题的执行次序现在依然可能，只不过，这一次，在执行次序的后期、f()之前，多了 auto_ptr 的构造函数。这是一个例子：

- (1) 为 T1 分配内存；
- (2) 构造 T1；
- (3) 为 T2 分配内存；
- (4) 构造 T2；
- (5) 构造 auto_ptr<T1>；
- (6) 构造 auto_ptr<T2>；
- (7) 调用 f()。

^⑩ 用作双关语。

上面的情况中，如果第(3)步或第(4)步中的任何一步抛出异常，同样的问题还是会发生在。类似地：

- (1) 为 T1 分配内存；
- (2) 为 T2 分配内存；
- (3) 构造 T1；
- (4) 构造 T2；
- (5) 构造 auto_ptr<T1>；
- (6) 构造 auto_ptr<T2>；
- (7) 调用 f()。

同样，如果第(3)步或第(4)步中的任何一步抛出异常，相同的问题还是会发生。

但幸运的是，这并非 auto_ptr 的罪过；在这里，我们是在以错误的方式使用 auto_ptr，仅此而已。一会儿，我们将看到几种使用 auto_ptr 的更好方式。

附笔：这不是个方案

注意，下面的做法也不是一个方案：

```
// 在某个头文件中：
void f( auto_ptr<T1> = auto_ptr<T1>( new T1 ),
        auto_ptr<T2> = auto_ptr<T2>( new T2 ) );

// 在某个实现文件中：
f();
```

这段代码为什么不是一个方案？因为它和“使用了表达式运算的例 21-1”是完全一样的。缺省参数是在函数调用表达式中创建的，即使它们完全写在别的地方（函数声明中）。

一个受限的方案

2. 演示如何写一个 `auto_ptr_new` 工具，用来解决问题 1 中的安全问题，并且可以通过如下方式调用：

```
// 例 21-2
//

// 在某个头文件中：
void f( auto_ptr<T1>, auto_ptr<T2> );

// 在某个实现文件中：
f( auto_ptr_new<T1>(), auto_ptr_new<T2>() );
```

最简单的方案是，提供一个像下面这样的函数模板：

```
// 例 21-2(a): 不完整的方案
//
template<typename T>
auto_ptr<T> auto_ptr_new()
{
    return auto_ptr<T>( new T );
}
```

这解决了异常安全问题。它产生的代码顺序不会导致资源泄漏，因为现在我们只有两个函数，而且我们知道，只有在一个函数完全执行结束之后，另一个函数才会执行。请看下面的运算次序：

- (1) 调用 `auto_ptr_new<T1>()`
- (2) 调用 `auto_ptr_new<T2>()`

如果第(1)步抛出异常，没有内存泄漏。因为 `auto_ptr_new()` 模板本身具有强烈的异常安全性。

如果第(2)步抛出异常，通过第(1)步创建的临时 `auto_ptr<T1>` 对象一定会被清除吗？是的，会。有人会奇怪：这不和条款 20 例 20-2 中相应情况下通过 `new T1` 产生的对象大同小异吗？那一个为什么没有被正确地清除呢？不，这一次大不一样！因为这里的 `auto_ptr<T1>` 实际上是个临时对象，而“消除临时对象”在标准中是有明文规定的。以下摘自标准第 12.2/3 节：

摧毁临时对象是“（从代码书写的角度来看）这个对象被创建时所在的整个表达式”运算的最后一步。即使那个表达式以抛出一个异常结束，也是如此。

但例 21-2(a)是一个有局限性的方案：它只对缺省构造函数有效，如果某个 `T` 没有缺省构造函数，或者你不想使用缺省构造函数，办法就行不通。还是得有一个更通用的方案。

使 `auto_ptr_new()` 方案更具通用性

正如 Dave Abrahams 所指出的那样，我们可以提供一组重载的函数模板来扩充我们的方案，使之支持非缺省构造函数：

```
// 例 21-2(b): 改进方案
//
template<typename T>
auto_ptr<T> auto_ptr_new()
{
    return auto_ptr<T>( new T );
}

template<typename T, typename Arg1>
```

```

auto_ptr<T> auto_ptr_new( const Arg1& arg1 )
{
    return auto_ptr<T>( new T( arg1 ) );
}

template<typename T, typename Arg1, typename Arg2>
auto_ptr<T> auto_ptr_new( const Arg1& arg1,
                           const Arg2& arg2 )
{
    return auto_ptr<T>( new T( arg1, arg2 ) );
}

// 其它

```

现在，`auto_ptr_new()`可以充分、自然地支持非缺省构造函数。

一个更好的方案

尽管 `auto_ptr_new()` 还不错，但有没有更好的方法可以让我们避免所有的异常安全问题，同时无需提供那样的辅助函数呢？我们能否通过更好的编程规范来避免这些问题呢？是的，这里就有一条可能的规范，它可以解决这个问题：

在一个表达式中，如果其它代码有可能抛出异常，就绝对不要在这个表达式中分配资源（例如，通过 `new`）。即使通过 `new` 获得的资源会立即在同一个表达式中被管理起来（例如，被传给一个 `auto_ptr` 的构造函数），也不要这样做。

对例 21-1 的代码来说，遵循这一规范的做法是，将其中一个临时 `auto_ptr` 对象放进一个独立的命名变量中：

```

// 例 21-1(a): 一个方案
//

// 某个头文件中
void f( auto_ptr<T1>, auto_ptr<T2> );

// 某个实现文件中
{
    auto_ptr<T1> t1( new T1 );
    f( t1, auto_ptr<T2>( new T2 ) );
}

```

这符合上面的规范。虽然我们还是在分配资源，但它不会因为某个异常而被泄漏，因为它不是在一个“有其它代码会抛出异常”的表达式中创建的^⑩。

这里还有另外一条可能的编程规范，它更简单、更容易做到（在阅读代码的时候更

^⑩ 在这儿我有意说得模糊一点，但也就一点。虽然 `f()` 的函数体包含在表达式运算之中，但我们不在意它是否会抛出异常。

容易让人理解)：在各自独立的程序语句中执行每一个显式的资源分配(例如, `new`)，并将(通过`new`)分配的资源立即交给管理者对象(例如, `auto_ptr`)。

在例 21-1 中，遵循第 2 条规范的做法是，将两个临时 `auto_ptr` 对象分别放进各自的命名变量中：

```
// 例 21-1(b): 一个更简单的方案
//

// 在某个头文件中:
void f( auto_ptr<T1>, auto_ptr<T2> );

// 在某个实现文件中:
{
    auto_ptr<T1> t1( new T1 );
    auto_ptr<T2> t2( new T2 );
    f( t1, t2 );
}
```

这符合第 2 条规范，我们不需要动一点脑筋就可以做到。新的资源都在各自的表达式中创建，然后被立即交给管理对象。

总结

我的建议是：

设计准则

在各自独立的程序语句中执行每一个显式的资源分配(例如, `new`)，并将(通过`new`)分配的资源立即交给管理者对象(例如, `auto_ptr`)。

这条准则易于理解和记忆，它巧妙地避免了最初提问中的所有异常安全问题，而且，通过使用管理者对象，它还帮助我们避免了其它许多异常安全问题。你可以考虑将这条准则加入到你们开发小组的编程规范中。

致谢

本条款启发于 `comp.lang.c++.moderated` 上的一个讨论专题。解答方案借鉴了 James Kanze、Steve Clamage 和 Dave Abrahams 在那个专题以及其它专题和私人信件中提供的分析结果。

条款 22：异常安全与类的设计，

之一：拷贝赋值

难度：7

举例来说，要想让任何一个 C++ 类的拷贝赋值运算符具有强烈的异常安全性，有可能做到吗？如果可能，如何做到？会有什么问题和后果？为了阐述这个议题，本条款介绍并解答了 Cargill Widget 范例。

- (1) 异常安全的三个常见级别是什么？简要地对它们进行说明，并解释它们的重要性。
- (2) 具有强烈异常安全性的“拷贝赋值”的规范形式是什么？
- (3) 请看下面这个类：

```
// 例 22-1: Cargill Widget 范例
//
class Widget
{
public:
    Widget& operator=( const Widget& ); // ???
    // ...
private:
    T1 t1_;
    T2 t2_;
};
```

假设 T1 或 T2 的某一操作会抛出异常。如果不改变类的结构，有可能写出一个具有强烈异常安全性的 Widget::operator=(const Widget&)吗？为什么可能？或者，为什么不可能？给出你的结论。

- (4) 说明并示范一种简单的转换技术：这种转换可以应用于任何一个类，并可以很容易地使得那个类的拷贝赋值具有（近似）强烈的异常安全性。在其它哪些场合下，我们看到过这种转换技术？

解答

本条款回答以下问题：

- 可以让任意一个类变得具有异常安全性吗——也就是说，无需改变它的结构？
 - 如果不行（也就是说，如果异常安全确实影响到类的设计），能否只做某种简单的改动，我们就总能让任意一个类变得具有异常安全性？
 - 在表达类之间的关系时，我们所选择的方式会带来哪些异常安全方面的后果？
- 具体来说，如果我们用继承或是委托来表达一个关系，对异常安全会有影响吗？

回顾：异常安全的规范形式

1. 异常安全的三个常见级别是什么？简要地对它们进行说明，并解释它们的重要性。

以下是规范的 Abrahams 保证（译注：此保证最初由 Dave Abrahams 提出，故名。请参阅 [sutter00] 条款 11. ）

- 基本保证（basic guarantee）：如果有异常抛出，资源不会泄漏，对象保持可摧毁、可使用但不一定可预测的状态。这是异常安全可以使用的最弱级别，它适用的场合是：失败的操作已经对对象状态做了改变，但调用者代码仍然能够应付。

- 强烈保证（strong guarantee）：如果有异常抛出，程序状态保持不变。这一级别总是蕴涵着“提交或回退（commit-or-rollback，意即付诸行动或恢复原状）”语义，这包括：如果操作失败，指向容器内部的引用或迭代器不会失效。

此外，某些函数只有提供更严格的保证，以上异常安全级别才有可能得以实现。

- 不抛出异常的保证（nothrow guarantee）：在任何条件下，函数都不会产生异常。有时，除非某些函数保证不抛出异常（例如，析构函数、资源释放函数），否则不可能实现强烈保证甚至基本保证。正如我们将要在下面看到的那样，标准 auto_ptr 的一个重要特性是，auto_ptr 操作决不会抛出异常。

2. 具有强烈异常安全性的“拷贝赋值”的规范形式是什么？

规范形式的拷贝赋值包括两步操作。首先，提供一个不抛出异常的 Swap() 函数，用以交换两个对象的内容，或内部状态：

```
void T::Swap( T& other ) /* throw() */
{
    // ... 交换*this 和 other 的内容 ...
}
```

接着，运用“创建一个临时对象然后交换”的手法来实现 operator=()：

```
T& T::operator=( const T& other )
{
    T temp( other ); // 先在一旁完成所有的工作,
    Swap( temp ); // 然后“提交”工作，仅使用
    return *this; // 不抛出异常的操作
}
```

分析 Cargill Widget 范例

这是 Tom Cargill 提出的有关异常安全问题的一个挑战：

3. 请看下面这个类：

```
// 例 22-1: Cargill Widget 范例
//
```

```

class Widget
{
public:
    Widget& operator=( const Widget& ); // ???
    // ...
private:
    T1 t1_;
    T2 t2_;
};

```

假设 T1 或 T2 的某一操作会抛出异常。如果不改变类的结构，有可能写出一个具有强烈异常安全性的 Widget::operator=(const Widget&)吗？为什么可能？或者，为什么不可能？给出你的结论。

简而言之：不改变 Widget 的结构一般无法达成异常安全。在例 22-1 中，根本不可能写出一个安全的 Widget::operator=()。如果有异常抛出，我们甚至不能保证 Widget 对象会最终处于某种调和状态，因为我们无法以原子操作的方式改变 t1_ 和 t2_ 成员的状态；或者，在中途出错的情况下，我们甚至不能可靠地回退到某种一致（相同或不同）的状态。假设 Widget::operator=() 准备首先更改 t1_，然后更改 t2_（对其中一个成员的操作必然在另一个之前，本例中，哪一个在前没有关系）。那么，问题具有双重性：

如果更改 t1_ 的操作抛出了异常，t1_ 必须保持不变。也就是说，Widget::operator=() 的异常安全性根本上依赖于 T1 提供的异常安全保证，即，T1::operator=()——或正在使用的任何其它“执行修改操作的函数（mutating function）”——要么得成功，要么不改变它的目标。这就几乎是在要求 T1::operator=() 提供强烈保证。同样的论证适用于 T2::operator=()。

如果更改 t1_ 的操作成功，但更改 t2_ 的操作抛出了异常，我们就处于“中途”状态，通常无法撤消对 t1_ 已做的修改。例如，如果“对 t1_ 重新赋以原始值或其它什么合理值”的操作也失败了呢？那么，Widget 对象甚至不一定能恢复到某种一致的状态，Widget 也就不能维持其不变性。

所以，只要 Widget 的结构像例 22-1 那样，我们就无法使它的 operator=() 具有强烈的异常安全性。（参看下面的副栏“一个简单一些但仍然有难度的 Widget”，这是一个稍微简单一些的例子，包含了以上问题的子集）

我们的目标是写一个具有强烈异常安全性的 Widget::operator=()，但对于 T1 或 T2 的所有操作，我们不能有异常安全方面的任何假设。可以做到吗？难道，完全无能为力了吗？

一个简单一些但仍然有难度的 Widget

可以看到，Cargill Widget 范例和下面这个简单一些的例子并非完全不同。

```
class Widget2
{
    // ...
private:
    T1 t1_;
};
```

即使是简化了的 Widget2，正文中的第一个问题还是存在。如果 T1::operator=()在已经开始修改目标的情况下还可以抛出异常，我们就无法写出一个具有强烈异常安全性的 Widget2::operator=()，除非 T1 通过其它某个函数提供了合适的功能。但如果 T1 可以那么做，我们为什么不让 T1::operator=()也那么做呢？

通用技术：使用 Pimpl 手法

4. 说明并示范一种简单的转换技术：这种转换可以应用于任何一个类，并可以很容易地使得那个类的拷贝赋值具有（近似）强烈的异常安全性。在其它哪些场合下，我们看到过这种转换技术？

幸运的是，尽管不改变 Widget 的结构就无法使得 Widget::operator=()具有强烈的异常安全性，但通过下面这个简单的转换手法，我们总可以实现一个具有“近乎”强烈异常安全性的赋值。即：通过“指针”而不是“值”来拥有成员对象，最好是运用 Pimpl 转换手法，将一切都隐藏在单个指针之后。（关于 Pimpl 手法的更详细介绍，包括使用它所带来的成本以及如何降低那些成本，请参见 *Exceptional C++* [Sutter00] 条款 26 至 30。）

例 22-2 演示了这个通用的“提升异常安全性”的转换手法（作为选择，pimpl_ 也可以用一个普通指针来保存，或者，你也可以使用其它某种指针管理对象）：

```
// 例 22-2: Cargill Widget
// 范例的通用解决方案
//
class Widget
{
public:
    Widget(); // 用新的 WidgetImpl 来初始化 pimpl_
    ~Widget(); // 这个析构函数必须提供，因为隐式
               // 生成的版本会导致使用上的问题
               // (参见条款 30 和 31)
```

```

Widget& operator=( const Widget& )
{
    // ...
private:
    class WidgetImpl;
    auto_ptr<WidgetImpl> pimpl_;
    // ... 提供可以正常工作的拷贝
    // 构造函数，或者禁止它 ...
};

// 然后，一般是在一个
// 单独的实现文件中：
//
class Widget::WidgetImpl
{
public:
    // ...
    T1 t1_;
    T2 t2_;
};

```

附笔：请注意，如果使用 `auto_ptr` 成员，那么：

- (a) 要么，你必须将 `WidgetImpl` 的定义提供给 `Widget`；要么，如果你想隐藏 `WidgetImpl`，你就必须为 `Widget` 写一个自己的析构函数，即使这个析构函数很简单^⑨；
- (b) 对于 `Widget`，你还应该提供自己的拷贝构造和赋值函数，因为一般来说，你不会希望类的成员具有“拥有权转移”语义。如果你有另外一种智能指针，你也可以考虑用它来取代 `auto_ptr`，但上述原则依然重要。

现在，我们可以很容易地实现一个不抛出异常的 `Swap()`，这意味着，我们可以很容易地写出一个“近似满足强烈异常安全保证”的拷贝赋值函数。首先，提供一个不抛出异常的 `Swap()` 函数，用以交换两个对象的内容（状态）。注意，这个函数确实能够提供不抛出异常的保证，即，在任何情况下它都不会有异常抛出，因为 `auto_ptr` 操作不允许抛出异常^⑩：

```

void Widget::Swap( Widget& other ) /* throw() */
{
    auto_ptr<WidgetImpl> temp( pimpl_ );
    pimpl_ = other.pimpl_;
    other.pimpl_ = temp;
}

```

^⑨ 如果使用编译器自动生成的析构函数，那个析构函数将被定义在每个编译单元（translation unit）中，因而，`WidgetImpl` 的定义必然在每个编译单元可见。

^⑩ 注意，如果将这个有三行代码的 `Swap()` 函数体用单行的 “`swap(pimpl_, other.pimpl_);`” 来代替，就不一定会工作正常；因为，对于 `auto_ptr`, `std::swap()` 不一定会工作正常。

然后，通过“创建一个临时对象然后交换”的手法实现 `operator=()`，从而达成 `operator=()` 常见的异常安全形式：

```
Widget& Widget::operator=( const Widget& other )
{
    Widget temp( other ); // 先在一旁完成所有的工作,
    Swap( temp );        // 然后“提交”工作，仅使用
    return *this;          // 不抛出异常的操作
}
```

这具有“近似”强烈的异常安全性。如果一个异常抛出，它不绝对保证程序状态会完全保持不变。你知道为什么吗？因为，当我们创建临时 `Widget` 对象并因此创建 `pimpl` 的 `t1_` 和 `t2_` 成员的时候，那些成员的创建（以及/或者析构——如果操作失败的话）可能会有副作用，比如修改一个全局变量，对于这一点，我们无法知晓，也无法控制。

潜在的异议及其不合理的原因

一些人会赶紧作出这样的结论，并带着战斗宣言般的口吻：“啊哈，这证明异常安全一般来说无法做到，因为不对类作修改，你就无法解决‘让任意一个类具有强烈异常安全性’这个一般性问题。”我之所以提到这一点，是因为的确有人提出过这一异议。

这个结论显然站不住脚。`Pimpl` 转换的确是这个一般性问题的解决之道，它只是对结构做了微小的改变。和大多数实现目标一样，异常安全也影响到类的设计。就是这样！譬如，要想使一个类具有多态性，你就得接受一些轻微的修改，让这个类从某个基类继承；同样，要想让一个类具有异常安全性，你就得接受一些轻微的修改，让这个类和它拥有的成员保持一定的距离。为作进一步说明，请看下面三条陈述：

- 无理陈述之一：C++中无法做到多态，因为对于任意一个类来说，如果你不去修改它（使之从基类 `Base` 继承），你就无法使得它可以在使用中代替 `Base&`。
- 无理陈述之二：C++中无法使用 STL 容器，因为对于任意一个类来说，如果你不去修改它（为它提供一个赋值运算符），你就无法将它用于一个 STL 容器。
- 无理陈述之三：C++中无法做到异常安全，因为对于任意一个类来说，如果你不去修改它（将其内部对象置于一个 `Pimpl` 类中），你就无法让它具有异常安全性。

以上论点全都是无效的。`Pimpl` 转换手法的确是一个普遍方案，利用它，即使对类数据成员的安全性一无所知，你也可以写出一个可以提供异常安全保证（的确，近似强烈保证）的类。

那么，我们可以得出哪些结论呢？

结论 1：异常安全影响到类的设计

异常安全绝对不只是“一个实现细节”。Pimpl 转换是一种简单直接的“结构上的改变”，但毕竟是个改变。

结论 2：总能让你的代码具有（近似）强烈的异常安全性

这里有一条重要的论断：

仅仅因为你所使用的类不具有起码的异常安全性，就说使用这个类的代码不能做到强烈的异常安全（副作用除外），这是没有道理的。

任何人可以使用一个缺少强烈异常安全的拷贝赋值运算符，并且可以在使用中做到强烈的异常安全；当然，如果 Widget 操作会产生副作用（例如修改一个全局变量），那就另当别论。因为对于这种情况，我们无从知晓也无法控制。换句话说，我们可以提供所谓的“局部强烈保证”：

局部强烈保证：如果有异常抛出，就“正在被操作的对象”而言，程序状态保持不变。这一级别总是意味着局部“提交或回退（commit-or-rollback）”语义，这包括：如果操作失败，指向容器内部的引用或迭代器不会失效。

对于这种“将细节隐藏在指针之后”的技术，Widget 的实现者和 Widget 的用户都可以做得很好。但如果是由 Widget 的实现者来做，它就总会是安全的，用户就不必做下面这样的事：

```
// 例 22-3：如果 Widget 的设计者
// 没有那样做，用户就必须得做
//
class MyClass
{
    auto_ptr<Widget> w_; // 通过一定的距离保存“拷贝
                        // 操作不安全”的 Widget
public:
    void Swap( MyClass& other ) /* throw() */
    {
        auto_ptr<Widget> temp( w_ );
        w_ = other.w_;
        other.w_ = temp;
    }
}
```

```

 MyClass& operator=( const MyClass& other )
{
    MyClass temp( other );      // 先在一旁完成所有的工作,
    Swap( temp );              // 然后“提交”工作，仅使用
    return *this;               // 不抛出异常的操作
}

// ... 提供可以正常工作的析构函数、
// 拷贝构造和赋值函数、
// 或者禁止它们 ...
};

```

结论 3：明智地使用指针

Scott Meyers 这么写道^①：

每当谈到 EH (Exception Handling, 异常处理)，我告诉大家两件事：

(1) **指针是你的敌人。因为它们带来的种种问题正是 `auto_ptr` 想要消除的。**

这就是说，普通指针通常应该被管理者对象拥有，这个管理者对象拥有所指向的资源，并自动执行清理工作。然后，Scott 继续写道：

(2) **指针是你的朋友。因为指针上的操作不会抛出异常。**

然后我祝大家天天快乐。

Scott 很好地运用二分法 (dichotomy) 捕捉到了指针的两面性。幸运的是，在实践中，你可以在夹缝中做到最好，也应该做到最好。

- 使用指针吧，因为它们是你的朋友，因为指针上的操作不会抛出异常。
- 和它们保持友好，将它们包装在 `auto_ptr` 这样的管理者对象中，因为这将保证正确的清理工作。这不会使指针“不抛出异常”的优点打折扣，因为 `auto_ptr` 操作同样绝不会抛出异常（而且只要你需要，你总可以得到 `auto_ptr` 内部真正的指针——例如，通过调用 `auto_ptr::get()`）。

通常，实现 Pimpl 手法的最好方式是前面例 22-2 中所演示的那样，它使用了一个指针（以利用操作不抛出异常的优点），同时还将动态资源安全地包装在一个管理者对象中（此例中，是一个 `auto_ptr`）。请记住，如果使用了 `auto_ptr`，你的类就必须提供自己的具有正确语义的析构函数、拷贝构造函数和拷贝赋值函数；或者，如果拷贝构造和赋值函数对你的类没有用，你可以禁止掉它们。

^① 摘自 Scott Meyers 的私人信函。

下一条款就“类之间的一种常见关系”展开了讨论，我们将运用以上所学，分析表达这一关系的最佳方式。

条款 23：异常安全与类的设计。

之二：继承

难度：6

Is-Implemented-In-Terms-Of 的含义是什么？如果告诉你，在选用继承和委托的时候，异常安全方面必然会影响到你，你可能会感到吃惊，你能指出会有哪些影响吗？

(1) Is-Implemented-In-Terms-Of 的含义是什么？

(2) 在 C++ 中，Is-Implemented-In-Terms-Of 可以通过非公有继承或包容/委托 (containment/delegation) 来表达。具体来说，在写一个类 T 的时候，如果它要用类 U 来实现，两个主要选择是：让 T 从 U 私有继承，或者，让 T 包含一个 U 成员对象。

在这些技术间所做的选择会和异常安全有什么牵连吗？请解释。（忽略和异常安全无关的主题）

解答

Is-Implemented-In-Terms-Of

创造了 Has-A, Is-A, Uses-A 这样的短句，在描述代码中的各种关系时，我们就有了一种方便的简写形式。

“Is-A（是一个）”——或者更准确地称为“Is-Substitutable-For-A（可以替换为一个）”——必须符合 Barbara Liskov 的可替换原则 (LSP, Substitutability Principle)，在这条原则中，Barbara 定义了“类型 S 可以替换为类型 T”的含义：

如果对于类型 S 的每一个对象 o1，都有类型 T 的一个对象 o2，使得在“用类型 T 定义的所有程序 P”中，o1 替换 o2 后 P 的行为不变，那么 S 是 T 的一个子类型 (subtype). [Liskov88]

所有的公有继承都应该符合 LSP，从而保证可替换性；Is-A 就常被用来描述这种公有继承。例如，“D Is-A B”的含义是，如果代码可以通过指针或引用使用基类 B 的对象，那么，它可以在使用中无缝地用“公有派生而来的子类 D 的对象”取代基类 B 的对象。

但要记住的重要一点，在 C++ 中有很多方式可以表示“Is-A”，比仅限于继承这一范围所能想象的要多。Is-A 还可以用来描述“支持相同接口但（从继承的角度来看）不

相关”的类，这些类可以相互替换地用于“具有公共接口的模板化代码”中。LSP 也适用于这种形式的可替换性，就像它适用于其它形式一样。例如，在这种形式下，“X Is-A Y”或“X Is-Substitutable-For-A Y”表示：如果模板化代码接受类型 Y 的对象，那么，它也会接受类型 X 的对象，因为 X 和 Y 都支持相同的接口。如果你对可替换性主题感兴趣，可以先从 Kevlin Henney 的一篇文章“Substitutability”[Henney00]开始。

很明显，这两种可替换性都依赖于对象实际使用的场合，但这里想要说明的要点是，Is-A 可以用不同的方式实现。正如我们即将看到的，Is-Implemented-In-Terms-Of 也是这样。

1. Is-Implemented-In-Terms-Of 的含义是什么？

代码中另一个同样常见的关系是 Is-Implemented-In-Terms-Of（用……来实现），或简称为 IIITO。如果 T 在它的实现中以某种形式使用了另一种类型 U，就称“T IIITO U”。 “以某种形式使用”这一措词当然留了很大的余地，它表示的范围很广。例如，T 可以是 U 的一个适配器（adapter）、代理（proxy）或包装类（wrapper）；或者，T 仅仅只是在它的实现细节中偶尔用到了 U。

“T IIITO U”通常意味着：要么，T 有一个 U（T Has-A U），如例 23-1(a)所示：

```
// 例 23-1(a): "T IIITO U", 运用 Has-A
//
class T
{
    // ...
private:
    U* u_; // 或者，通过“值”或“引用”
};
```

要么，T 非公有派生于 U，如例 23-1(b)所示^②：

```
// 例 23-1(b): "T IIITO U", 运用派生
//
class T : private U
{
    // ...
};
```

很自然地，这给我们带来以下问题：如果可以选择，哪种方式实现 IIITO 会更好？两种方式各有什么优劣？应该分别在何时考虑使用它们中的每一个？

^② 是的，公有派生也可以表达 IIITO，但公有派生的主要含义还是 Is-Substitutable-For-A。

如何实现 IIITO：继承还是委托？

2. 在 C++ 中，*Is-Implemented-In-Terms-Of* 可以通过非公有继承或包容/委托（*containment/delegation*）来表达。具体来说，在写一个类 T 的时候，如果要用类 U 来实现，两个主要选择是：让 T 从 U 私有继承，或者，让 T 包含一个 U 成员对象。

正如我以前曾经指出的那样，继承常被使用过度，甚至有经验的程序员也是如此。一条有效的工程设计原则是：将耦合性降至最低。如果一种关系（relationship）可以用多种有效方式来表达，请使用关系最弱（松散）的那一个。既然继承近乎是 C++ 中可以表达的最强烈（紧密）的关系——仅次于友元^⑩，那么，只有在没有更弱的关系可供选择时，使用它才真正合适。如果仅用委托就可以表达出类的关系，应该总是优先选择委托。

最小耦合性原则无疑会直接影响到代码的健壮性（或脆弱性）、编译时间，以及其它可见后果。有趣的是，为了实现 IIITO，在继承和委托之间所做的选择还会有异常安全上的牵连。一会儿你将看到，最小耦合性原则也和异常安全有关，这一点不应该让人感到惊奇；因为在一个设计中，耦合性对它“可能具有的异常安全性”有直接影响。

耦合性原则是这样描述的：

低耦合性提高程序的正确性（包括异常安全），高耦合性降低程序的“最大可能的正确性”（包括异常安全）。

这再自然不过。毕竟，在现实世界中，对象之间的关系越松散，它们之间的相互影响就越小。正因为如此，我们才会在建筑物中建防火墙，在船舱之间装上隔板。如果一个船舱出了问题，船舱隔离得越多，在事情得以控制之前，问题就越不容易扩散到其它船舱。

现在，让我们回到例 23-1(a) 和 23-1(b)，再次考虑那个“用另一个类型 U 来实现类 T”的问题。看看拷贝赋值运算符：要想表达 IIITO 关系，我们所做的选择将会怎样影响到 T::operator=()？

对异常安全的影响

在这些技术间所做的选择会和异常安全有什么牵连吗？请解释。（忽略和异常安全无关的主题）

^⑩ 对一个类 X 来说，友元与之具有最大可能的紧密关系，因为友元可以访问并依赖于 X 的所有成员。从 X 派生的类仅可以访问并依赖于 X 的公有和保护成员。

首先来看看，如果想用 Has-A 来表达 IIITO 关系，我们应当怎样写出 T::operator=()。我们当然会遵循好的习惯，运用“先在一旁完成所有工作，然后只使用不抛出异常的操作来提交工作”这一常用技术，从而将异常安全性提升至最高，所以我们会像下面这样写：

```
// 例 23-2(a): “T IIITO U”，使用 HAS-A
//
class T
{
    // ...
private:
    U* u_;
};

T& T::operator=( const T& other )
{
    U* temp = new U( *other.u_ ); // 先在一旁完成所有工作

    delete u_; // 然后“提交”工作，仅使用
    u_ = temp; // 不抛出异常的操作
    return *this;
}
```

很不错。不需要对 U 做任何假设，我们可以写出一个具有“近似”强烈异常安全性的 T::operator=()，当然，U 可能产生副作用的情况除外。

即使 U 对象本来是作为“值”而不是作为“指针”包含在 T 中，我们也可以很容易地将其转换为通过指针来保存，就像上面那样。我们还可以将 U 对象放进一个 Pimpl 中；这种使用了转换技术的 Pimpl 手法，本短系列的第一部分做过介绍。可见，委托（Has-A）给了我们很大的灵活性，它让我们可以很容易地写出一个具有一定异常安全性的 T::operator=()，但无需对 U 做任何假设。

再来看看，一旦 T 和 U 之间的关系涉及到任何方式的继承，问题会有什么变化：

```
// 例 23-2(b): “T IIITO U”，使用派生
//
class T : private U
{
    // ...
};

T& T::operator=( const T& other )
{
    U::operator=( other ); // ???
    return *this;
}
```

问题出在对 U::operator=()的调用。正如条款 22 副栏中提到的那样（谈论的是类似的情况），如果 U::operator=()可以在“已经开始修改目标的情况下”抛出异常，我们就无法写出一个具有强烈异常安全性的 T::operator=()，除非 U 通过其它某个函数提供了合适的功能。（但如果 U 可以那么做，为什么不让 U::operator=()也那么做呢？）

换句话说，现在，T 为它的成员函数 T::operator=()提供异常安全保证的能力必然依赖于 U 的安全和保证。再次想想，这应该让人感到奇怪吗？不。因为，在表达 T 和 U 之间的联系时，例 23-2(b)使用了最大可能的紧密关系，从而导致了最大可能的高耦合性。

总结

松散的耦合促进程序的正确性（包括异常安全），紧密的耦合降低程序的“最大可能的正确性”（包括异常安全）。

继承常被过度使用，即使是有经验的程序员也是如此。请阅读 *Exceptional C++* [Sutter00] 条款 24，那个条款针对“应当尽可能地使用委托而不使用继承”的原因和做法进行了分析，它涉及了包括异常安全在内的许多内容。无论何时都要做到将耦合性降至最低。如果类的关系可以用多种方式表达，请使用关系最弱的那个有效方式。特别是，只有在委托不能独立完成使命情况下，我们才会使用继承。

继承与多态

不来点继承和多态，面向对象将会怎样？

尽管继承常被滥用，但它还是一种很重要的工具——这包括多继承。特别是，当你生活在现实世界中时，你会发现，你经常需要将不同供货商提供的程序库结合起来使用，此时，多继承便凸显它的价值。本章向你展示，在结合使用不同供货商提供的“基于继承”的程序库时，应当如何避免连体双胞（Siamese Twin）问题。此外，本章还示范了许多合理（以及一些不合理地）使用纯虚函数的方式，以及如何编写多继承的替代方案、如何对使用继承关系的用户施加控制。

条款 24：为什么要使用多继承？

难度：6

一些语言，包括 SQL99 标准，还在为“是否应该支持单继承或多继承”的问题大伤脑筋。本条款邀请您讨论这一主题。

(1) 什么是多继承 (MI，即 multiple inheritance)？在 C++ 中引入 MI 带来了哪些额外的可能性和复杂性？

(2) MI 究竟有必要吗？如果必要，尽可能多地列举出它的使用场合，并论证为什么应该将 MI 加入到语言中；如果不必要，请论证为什么单继承 (SI)（并且，可能结合 Java 风格的接口）可以取代多继承，甚至比它更出色，以及为什么不应该将 MI 加入到语言中。

解答

1. 什么是多继承 (MI，即 multiple inheritance)？在 C++ 中引入 MI 带来了哪些额外的可能性和复杂性？

非常简要地回答：MI 指的是从多个（多于一个）直接基类（direct class）继承的能力。

例如：

```
class Derived : public Base1, private Base2
{
    //...
};
```

在 C++ 中引入 MI 所带来的可能性是：一个类的同一个（直接或间接）基类（base class）可能会不只一次地作为它的基础类（ancestor）出现。这里有一个简单的例子，即那个经典的钻石形状的继承图，如图 4 所示。

这里，B 两次作为 D 的间接基类（indirect class）出现，一次是通过 C1，另一次则是通过 C2。

这种情况下，就很有必要引入 C++ 的另一个特性：虚拟继承。现在的问题是：程序员希望 D 拥有基类 B 的一个子对象还是两个？如果答案是一个，B 就应该是一个虚拟基类，图 4 就成为了可怕的死亡钻石。如果答案是两个，B 就应该是一个普通（非虚拟）基类。

最后，虚拟基类的主要复杂性在于：它们必须通过最底层的派生类（most-derived class）直接初始化。关于这一点的详细介绍，以及 MI 其它方面的知识，请参阅 [stroustrup00]，或[Meyers97]条款 43。

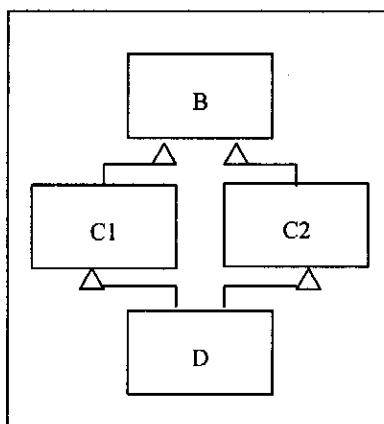


图 4：可怕的死亡钻石（如果 D 从 B 虚拟继承）

设计准则

避免多继承自多个“非 protocol 类”。（protocol 类是一种抽象基类（abstract base class），或简称 ABC，它完全由纯虚函数组成，没有数据成员。）

2. MI 究竟必要吗？

简短的回答是：只要程序可以用汇编语言（或更低级的语言）来写，就不能说某种特性绝对“必要”。然而，正如大多数人不会去用简单的 C 编写自己的虚函数机制一样，在某些场合下，没有 MI 会让你大费周章。

正因为如此，我们现在才有了这一被称为 MI 的神奇特性。但问题是——或者，至少前面的问题应该换为——MI 是个好东西吗？^⑨

简而言之，的确有人认为 MI 是个坏主意，因而无论如何都应该避免它。这不对。是的，如果你在使用 MI 时有欠考虑，它的确会招致不必要的耦合性和复杂性。然而，任何一种被误用的继承都是这样（参见 *Exceptional C++* [Sutter00] 条款 24），但我相信，大家不会就此认为继承不是个好东西。还有，是的，在完全不使用 MI 的情况下，任何程序都可以写出来，但你也得知道，在完全不使用继承的情况下，任何程序也都可以写出来。实际上，任何程序都可以用汇编语言来写。但这并不是说用汇编语言来写程序就一定是个好主意，相反，你甚至不愿意去做那样的事。

MI 如果必要，尽可能多地列举出它的使用场合，并论证为什么应该将 MI 加入到语言中；如果不必要，请论证为什么单继承（SI）（并且，可能结合 Java 风格的接口）可以取代多继承、甚至比它更出色，以及为什么不应该将 MI 加入到语言中。

那么，什么时候使用 MI 才算合适？简而言之，只有在每一个继承单独取出来看都合适的时候，这样的 MI 才算合适。*Exceptional C++* 条款 24 提供了一个相当详尽的列表，说明了什么时候该使用继承。在现实世界中，MI 的应用大多逃不出以下三类。

(1) **结合使用程序模块或程序库。**之所以首先提出这一点有一个理由，我将在后面说明。很多类被设计为基类——即，在使用时你得从它继承。这很自然地带来一个问题：如果你想写一个类，它用到了两个程序库，但每个程序库都要求你从它的某个类继承，这时候你该怎么办？

^⑨ 一定程度上，本条款的主题启发自 1998 年 6 月 SQL 标准会议上发生的事件，在那次会议中，多继承被从 ANSI SQL99 标准草案中删除了。（如果说得远一点的话，你们当中如果有人对数据库感兴趣，就会知道，在 SQL4 中 MI 又以修订形式重新恢复了。）当时之所以那样做，主要是因为所提出的多继承规范存在技术上的困难，并且是为了想和 Java 这些不真正支持多继承的语言看齐。另外，仅仅是坐在那儿听人们在那样一个相对太迟的时间里争论多继承的优缺点，这本身就让人觉得有趣。在 C++ 世界中，自从这种语言形成以来，我们很少做这种事；这让我回忆起几年前（或比这更近的时期）新闻组上广泛展开的激烈争论，我会忍不住自言自语地念出一些标题，譬如：“MI 是罪恶！！！”

在面对这种情况时，你一般无法通过修改程序库代码来避免某个继承。因为，它可能是从第三方供货商购买的程序库，或者，可能是你的公司里另一个项目组开发的模块。无论哪种情况，你不但不能修改代码，你甚至可能没有代码！如果是这样，MI 就是必要的；没有其它（自然的）方法可以帮助你去做你必须做的事；此时，使用 MI 完全合理。

在实践中我发现，知道如何运用 MI 来结合使用供应商提供的程序库，这是每一个 C++程序员必须具备的素质。无论你是否经常运用它，你绝对应该知道它并理解它。

(2) *Protocol 类 (interface 类)*. 在 C++中，MI 最合适、最安全的应用是定义 protocol 类——即，完全由纯虚函数构成的类。由于这种基类没有数据成员，MI 臭名昭著的复杂性就得以完全避免。

有趣的是，有的语言（或模型）通过非继承机制来支持这种 MI. Java 和 COM 就是两个例子。严格来说，Java 有多继承，但它将实现（implementation）的继承仅限于单继承。一个 Java 类可以实现多个“接口”，这种接口非常类似 C++中没有数据成员的纯抽象基类。COM 本身没有继承的概念（虽然在用 C++来写 COM 对象时，这是一种常用的实现技术），但它同样也有“接口组合（composition of interfaces）”的思想，COM 接口类似 Java 接口和 C++模板的结合。

(3) 易于（多态）使用. 有了继承，在接受基类对象的任何代码中，我们就都可以使用派生对象，这是一项威力强大的功能。在某些场合，如果同一个派生对象可以代替数种基类对象使用，那将会很有用处，这正是 MI 大显身手的地方。关于这一点有一个好例子，请参阅[Stroustrup00]14.2.2 节，那里演示了一个基于 MI 的设计，用以构造异常类（exception class）；在那个设计中，最底层的派生异常类可能和多个直接基类具有多态式的 Is-A 关系。

注意，第(3)点在很大程度上与第(1)、(2)点重叠。在实施另两点之一时，同时、并出于相同理由运用第(3)点，往往很有用处。

还要考虑到另一点，不要忘记：有时候，单纯从两个不同的基类继承并没有必要，相反，我们要让每一个继承都有不同的理由。“多态的 LSP Is-A 公有继承”并非唯一故事（译注：参见条款 23）；使用继承还有很多其它可能的原因。例如，一个类可能需要从一个基类 A 私有继承，以获得对类 A 的保护成员的访问权，同时，它还要从另一个基类 B 公有继承，以多态地实现类 B 的某个虚函数。

条款 25：模拟多继承

难度：5

如果不能使用多继承，如何模拟之？在 C++ 中，多继承为什么要以那样的方式运作？

这个练习将让你切身体会到其中一些原因。在你的答案中，你模拟出来的结果在句法上一定要尽可能地让调用者感到自然。

请看下面的例子：

```
class A
{
public:
    virtual ~A();
    string Name();
private:
    virtual string DoName();
};

class B1 : virtual public A
{
    string DoName();
};

class B2 : virtual public A
{
    string DoName();
};

A::~A() {}
string A::Name()    { return DoName(); }
string A::DoName() { return "A"; }
string B1::DoName() { return "B1"; }
string B2::DoName() { return "B2"; }

class D : public B1, public B2
{
    string DoName() { return "D"; }
};
```

不运用 MI，写一个与上面等价（或尽可能接近）的类 D，即，在不使用多继承的情况下，演示出你能找到的“拐弯抹角解决问题”的最好方法。要让 D 得到相同的效果和可用性，同时又要尽量避免对调用者代码的句法多做修改，如何做到？

* * * * *

起步：请先考虑下面测试程序中的各种情况，然后开始设计。

```

void f1( A& x ) { cout << "f1:" << x.Name() << endl; }
void f2( B1& x ) { cout << "f2:" << x.Name() << endl; }
void f3( B2& x ) { cout << "f3:" << x.Name() << endl; }

void g1( A x ) { cout << "g1:" << x.Name() << endl; }
void g2( B1 x ) { cout << "g2:" << x.Name() << endl; }
void g3( B2 x ) { cout << "g3:" << x.Name() << endl; }

int main()
{
    D d;
    B1* pb1 = &d;      // D* -> B* 转换
    B2* pb2 = &d;
    B1& rb1 = d;      // D& -> B& 转换
    B2& rb2 = d;

    f1( d );          // 多态
    f2( d );
    f3( d );

    g1( d );          // 切割
    g2( d );
    g3( d );
    // dynamic_cast/RTTI
    cout << ( (dynamic_cast<D*>(pb1) != 0) ? "ok " : "bad " );
    cout << ( (dynamic_cast<D*>(pb2) != 0) ? "ok " : "bad " );

    try
    {
        dynamic_cast<D&>(rb1);
        cout << "ok ";
    }
    catch(...)
    {
        cout << "bad ";
    }

    try
    {
        dynamic_cast<D&>(rb2);
        cout << "ok ";
    }
    catch(...)
    {
        cout << "bad ";
    }
}

```

解答

```
class D : public B1, public B2
{
    string DoName() { return "D"; }
};
```

不运用 MI，写一个与上面等价（或尽可能接近）的类 D，即，在不使用多继承的情况下，演示出你能找到的“拐弯抹角解决问题”的最好方法。要想让 D 得到相同的效果和可用性，同时又要尽量避免对调用者代码的句法多做修改，如何做到？

各种方案都有其不足之处，下面这一个和我们的要求十分接近。

```
class D : public B1
{
public:
    class D2 : public B2
    {
public:
    void Set ( D* d ) { d_ = d; }
private:
    string DoName();
    D* d_;
    } d2_;
};

D()           { d2_.Set( this ); }

D( const D& other ) : B1( other ), d2_( other.d2_ )
{ d2_.Set( this ); }

D& operator=( const D& other )
{
    B1::operator=( other );
    d2_ = other.d2_;
    d2_.Set(this);
    return *this;
}

operator B2&()     { return d2_; }

B2& AsB2()        { return d2_; }

private:
    string DoName()      { return "D"; }
};

string D::D2::DoName() { return d_->DoName(); }
```

继续往下阅读之前，先分析一下这段代码，体会每一个类或函数的用途。

不足

这个方案非常漂亮地实现了 MI，模拟了 MI 的大多数行为；只要你依照编程规范填补上没有完全实现的部分，它就可以支持 MI 的所有应用。下面特别列出了这个方案的一些缺点，说明了 MI 的哪一部分特性没有完全实现。

- 由于提供了 `operator B2&()`，较之指针，引用就得特殊（不一致地）对待。
- 将 D 作为 B2 使用时，调用者代码必须显式地调用 `D::AsB2()`。（这意味着，在测试程序中，必须将“`B2* pb2 = &d;`”修改为“`B2* pb2 = &d.AsB2();`”）
- 不能通过 `dynamic_cast` 将 `D*` 转换为 `B2*`。（如果不注意使用预处理，你可以重新定义 `dynamic_cast`，就有可能解决这一问题，但那是一个万不得已的方案）

有趣的是，你可能已经发现：和多继承所给出的内存布局相比，D 对象的内存布局与之类似。这是因为，我们是在尽力模拟 MI——虽然我们没有利用语言本身所提供的语法上的所有好处和便利。

你可能不是经常需要 MI，但当你需要它的时候，你是“真的”需要。本条款在于说明：对于这种有用的特性，如果语言提供了必要的支持，就远比你自起炉灶要好得多——即使你可以结合其它语言特性和编程规范完全模拟出它的功能。

条款 26：多继承与连体双胞问题

难度：4

对继承而来的函数进行改写很容易，但如果是在两个基类中有相同原型”的虚函数，情况就不一样。即使基类并非来自不同的软件供应商，这种情况也会发生！分割这种“连体双胞”（Siamese Twin）“函数的最佳方式是什么？

请看下面两个类：

```
class BaseA
{
    virtual int ReadBuf( const char* );
    // ...
};

class BaseB
{
```

```

virtual int ReadBuf( const char* );
// ...
};

```

BaseA 和 BaseB 有一个共同点——它们显然都想被用作基类，但除此之外它们毫不相干。它们的 ReadBuf() 函数用来做不同的事，而且这两个类还来自于不同的程序库供应商。

示范如何写一个 Derived 类，这个类从 BaseA 和 BaseB 公有继承，而且还要对两个 ReadBuf() 进行改写，让它们做不同的事。

解答

本条款指出了 MI 可能会带来的一个小毛病，并演示如何有效地处理它。假设在同一个工程中，你使用了两个供货商提供的程序库。供货商 A 提供了下面的基类 BaseA。

```

class BaseA
{
    virtual int ReadBuf( const char* );
    // ...
};

```

BaseA 的本意是希望其它类从它继承，并有可能改写它的某些虚函数；因为，程序库的其它部分也都希望，它们的对象可以被多态地作为 BaseA 来使用。这是一种很普遍很正常的做法，没有任何不对之处；对那些可扩充的应用程序框架而言，情况更是如此。

是的，一切都很正常。但当你开始使用供货商 B 的程序库时，你会坐立不安、惊诧万分：

```

class BaseB
{
    virtual int ReadBuf( const char* );
    // ...
};

```

“哦，真是太巧了！”你会想。在供货商 B 提供的程序库中，不仅也有一个基类需要你继承，而且这个基类也有一个虚函数，其原型 (signature) 刚好与类 BaseA 中的一个虚函数完全相同。然而，BaseB 中的那个函数打算做的是完全不同的事，这是问题的关键所在。

BaseA 和 **BaseB** 有一个共同点——它们显然都想被用作基类，但除此之外它们毫不相干。它们的 **ReadBuf()** 函数用来做不同的事，而且这两个类还来自于不同的程序库供货商。

示范如何写一个 **Derived** 类，这个类从 **BaseA** 和 **BaseB** 公有继承，而且还要对两个 **ReadBuf()** 进行改写，让它们做不同的事。

当你必须写一个“同时从 **BaseA** 和 **BaseB** 继承”的类时，问题就变得明显了。你之所以必须这样做，也许是因为你需要一个对象：对于这个对象，两个程序库中的函数都能够多态地使用它。下面就是针对这样一个函数的幼稚尝试：

```
// 例 26-1: 第 1 个尝试, 不正确
//
class Derived : public BaseA, public BaseB
{
    // ...
    int ReadBuf( const char* );
    // 同时改写 BaseA::ReadBuf()
    // 和 BaseB::ReadBuf()
};


```

这里，**Derived::ReadBuf()** 同时改写了 **BaseA::ReadBuf()** 和 **BaseB::ReadBuf()**。这不符合我们的要求，请看下面的代码：

```
// 例 26-1(a): 反例,
// 为什么第 1 个尝试不正确
//
Derived d;
BaseA* pba = &d;
BaseB* pbb = &d;

pba->ReadBuf( "Sample buffer" );
    // 调用 Derived::ReadBuf

pbb->ReadBuf( "Sample buffer" );
    // 调用 Derived::ReadBuf
```

看出问题来了吗？**ReadBuf()** 在两个接口中都是虚函数，它以多态的方式工作，这一点正是我们所期望的。但是，不管我们使用的是哪一个接口，被调用的都是同一个 **Derived::ReadBuf()** 函数。然而，**BaseA::ReadBuf()** 和 **BaseB::ReadBuf()** 具有不同的语义，它们打算做的是不同的事，而不是相同的事。进一步来看，对 **Derived::ReadBuf()** 来说，它无法分辨出自己被调用时是通过 **BaseA** 的接口，还是通过 **BaseB** 的接口（如果二者之一被调用的话）；所以，我们无法在 **Derived::ReadBuf()** 中加上“if”语句，使它可以根据不同的调用的方式做不同的事。这很让人伤脑筋，但我们必须面对。

“得啦！”你会想，“这完全是个凭空臆造的例子，对不对？”答案是：不对。例如，Microsoft 的 John Kdilin 曾经报告说，从 **IOleObject** 和 **IConnectionPoint** 这两个 COM 接口（可以将它们看作完全由公有虚函数构成的抽象基类）同时派生出来的类会有问题，因为：

(a) 这两个接口都有一个被说明为 virtual HRESULT Unadvise(unsigned long); 的成员函数；

(b) 通常，你必须改写每一个 Unadvise()，让它做不同的事。

暂停一下，好好想想这个例子。应该如何解决问题？有没有办法分别改写这两个继承而来的 ReadBuf 函数，使得每一个函数可以完成不同的功能呢？也就是说，根据外部代码在调用时是通过 BaseA 还是 BaseB 的接口，可以分别执行正确的操作？简而言之，如何分割这一连体双婴？

如何分割连体双婴？

幸运的是，有一个非常干练的解决方案。这个问题的关键在于，这两个可以改写的函数具有完全相同的名称和原型。所以，解决方案的关键必然在于：至少需要修改其中一个函数的原型，而函数原型中最容易修改的部分是函数名称。

如何改变一个函数的名称？当然是通过继承！这就需要一个中间类（intermediate class），这个中间类从基类派生，它说明一个新的虚函数，并改写那个继承而来的虚函数，使之调用这个新函数。继承层次结构看起来像图 5 中那样。

代码大致像这样：

```
// 例 26-2: 第 2 个尝试, 正确
//
class BaseA2 : public BaseA
{
public:
    virtual int BaseAReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p ) // 改写继承而来的版本
    {
        return BaseAReadBuf( p ); // 调用新函数
    }
};
```

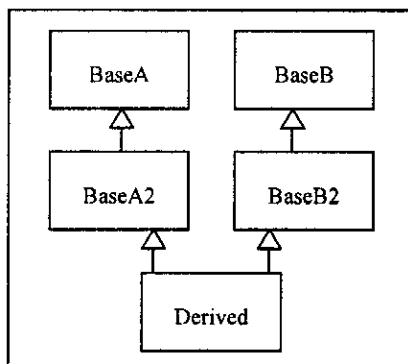


图 5：使用中间类，为继承而来的虚函数更名

```

class BaseB2 : public BaseB
{
public:
    virtual int BaseBReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p ) // 改写继承而来的版本
    {
        return BaseBReadBuf( p ); // 调用新函数
    }
};

class Derived : public BaseA2, public BaseB2
{
/* ... */

public: // 或者是“private:”，这取决于
// 别的代码是否应该直接调用它们

    int BaseAReadBuf( const char* );
    // 通过 BaseA2::BaseAReadBuf
    // 间接改写 BaseA::ReadBuf

    int BaseBReadBuf( const char* );
    // 通过 BaseB2::BaseBReadBuf
    // 间接改写 BaseB::ReadBuf
};

```

BaseA2 和 BaseB2 可能还需要复制 (duplicate) BaseA 和 BaseB 的构造函数，使得 Derived 可以调用到它们。但要做的修改也就这么多。(通常，比“在代码中复制构造函数”更简单的方法是：让 BaseA2 和 BaseB2 从基类虚拟继承。这样，Derived 就可以直接访问基类的构造函数。) BaseA2 和 BaseB2 是抽象类，所以，它们不需要复制 BaseA 或 BaseB 的其它任何函数或运算符，例如赋值运算符。

现在，一切都运作正常。

```

// 例 26-2(a): 为什么第 2 个尝试是正确的
//
Derived d;
BaseA* pba = d;
BaseB* pbb = d;

pba->ReadBuf( "Sample buffer");
// 调用 Derived::BaseAReadBuf

pbb->ReadBuf( "Sample buffer");
// 调用 Derived::BaseBReadBuf

```

对更底层的派生类来说，它们只需知道：一定不要再次改写 ReadBuf。如果它们真的那样做的话，我们在中间类中所做的一切努力（更改函数名称）都会付之东流。

条款 27: (非) 纯虚函数

难度: 7

为纯虚函数提供函数体究竟有没有意义?

(1) 什么是纯虚函数? 给出一个例子。

(2) 在说明一个纯虚函数后, 你为什么还会为它提供定义(函数体)? 尽可能多地列举出这样做的理由和场合。

解答

1. 什么是纯虚函数? 给出一个例子。

纯虚函数 (pure virtual function) 首先是一个虚函数, 此外, 它强迫实体派生类 (concrete derived class) 必须对它进行改写。一个类如果存在任何未改写的纯虚函数, 这个类就是一个“抽象”类, 你就不能创建那种类型的对象。

```
// 例 27-1
//
class AbstractClass
{
    // 说明一个纯虚函数:
    // 这个类现在是一个抽象类
    virtual void f(int) = 0;
};

class StillAbstract : public AbstractClass
{
    // 没有改写 f(int),
    // 所以这个类还是一个抽象类
};

class Concrete : public StillAbstract
{
public:
    // 最后改写了 f(int),
    // 所以这个类是一个实体类
    void f(int) /*...*/
};

AbstractClass a;      // 错误, 抽象类
StillAbstract b;     // 错误, 抽象类
Concrete      c;     // 正确, 实体类
```

2. 在声明一个纯虚函数后，你为什么还会为它提供定义（函数体）？尽可能多地列举出这样做的理由和场合。

你之所以会这样做，一般出于三个主要原因。在这三个原因中，第一个可谓老生常谈；第二个和第三个很有用处，但有点少见。第四个则是高级程序员偶尔使用的手段，用来应付功能较弱的编译器。（既然列出了三个原因，我们当然还需要第四个，这出于一条不言而喻的理由，即，对任何一个三部曲来说，没有第四部就是不完整的。⁽³⁾ ⁽⁴⁾）

1) 纯虚析构函数

所有基类的析构函数要么应该是虚拟公有成员，要么应该是非虚拟保护成员。简单地说，这是因为：首先，要记住，你应该总是避免从实体类（concrete class）派生。因而，假设基类不是实体类，那么，它就不会出于“实例化自身”的目的而去提供一个公有析构函数。这样，你就只剩下两个选择：

(a) 要么，你需要“通过基类指针进行多态删除”的功能，这种情况下，析构函数必须是虚拟公有成员；

(b) 要么，你不需要这一功能，这种情况下，析构函数应该是非虚拟保护成员——之所以是保护成员，是为了防止被滥用。更多介绍参见[Sutter01]。

如果一个类应该是抽象类（你想禁止将其实例化），但它没有其它任何纯虚函数，而有一个公有析构函数，那么，请将这个析构函数声明为纯虚函数（如上段所述，无论如何它也应该这样）。

```
// 例 27-2 (a)
//
// 文件 b.h
//
class B
{
public: /*...其它...*/
    virtual ~B() = 0; // 纯虚析构函数
};
```

当然，任何派生类的析构函数必须隐式地调用基类的析构函数，所以，析构函数还是得定义（即使为空）。

```
// 例 27-2 (a)，继续
//
// 文件 b.cpp
//
B::~B() { /* 可能为空 */ }
```

⁽³⁾ “我们的主要武器是惊讶、恐惧和惊讶是我们的两个主要武器。⁽⁴⁾”（摘自 Monty Python's Flying Circus）
(译注：Monty Python's Flying Circus 是著名的幽默短剧。)

如果不提供这个定义，你还是可以让其它类从 B 派生，但那些派生类就不能被实例化，从而没有什么特别的用处。

设计准则

基类的析构函数要么是虚拟公有成员，要么是非虚拟保护成员。

2) 明确使用缺省行为

如果派生类没有改写某个普通的虚函数，它就会默认地继承基类中的行为。如果想提供一个默认行为但又不想让派生类这么“无声无息”地继承，你可以声明一个纯虚函数并依然提供缺省实现；这样，派生类的设计者如果想使用它，就必须主动对它进行调用。

```
// 例 27-2(b)
//
class B
{
protected:
    virtual bool f() = 0;
};

bool B::f() {
    return true;           // 这是一个不错的缺省实现，但不
}                           // 应该被派生类稀里糊涂地使用

class D : public B
{
    bool f()
    {
        return B::f();    // 如果 D 需要缺省行为，就
    }                      // 必须像这样明确地说出来
};
```

在 Gamma 等四人所著的 *Design Patterns*[Gamma95]一书中，State 模式就是这样一个例子，它演示了如何将这一技术加以有效地利用。

3) 提供部分行为

有时，我们需要向派生类提供“部分行为”（partial behavior），同时，这个派生类还必须保持完整。这是一种很有价值的应用。其设计思想是：在派生类中，将基类实现作为派生类实现的“一部分”来执行。

```

// 例 27-2 (c)
//
class B
{
protected:
    virtual bool f() = 0;
};

bool B::f()
{
    // 执行某一通用操作
}

class D : public B
{
    bool f()
    {
        // 首先，使用基类的实现
        B::f();

        // ...现在，做其它工作
    }
};

```

再次参考[Gamma95]，Decorator 模式演示了如何有效地运用这种技术。

4) 应付功能不足的编译器诊断程序

有些情况下，你会无意中调用了一个纯虚函数（间接地从基类的构造函数或析构函数调用；具体例子请参阅你最常用的高阶 C++ 书籍）。当然，如果代码写得好，通常不会出这些问题，但任何人都可能聪明一世，糊涂一时。

不幸的是，在出现这种问题时，并不是所有的编译器^⑩都会确切地告诉你问题所在。一些编译器会给你一些词不达意的错误信息，让你永无止境地去追踪这个错误。“天啊！”几个小时后，当你最后终于诊断出错误所在时，你会尖叫：“我做了这种事，编译器为什么不早告诉我呢？”（回答：因为这是“不可预测行为”的表现之一。）

要想避免这种调试时间上的浪费，一个办法是：为这种永远不应该被调用的纯虚函数提供定义，并在那些定义中置入一些恶性代码。这样一来，如果你无意中调用了那些函数，你就会立刻知道。

例如：

^⑩ 是的，从技术上来说，这类错误由运行时环境来捕捉。但我还是要说编译器，因为，通常来说，这种在运行时为你检查错误的代码应该由编译器生成。

```

// 例 27-2(d)
//
class B
{
public:
    bool f();      // 可能会提供这个函数，调用 do_f()
                   // (非虚拟 Interface 模式)
private:
    virtual bool do_f() = 0;
};

bool B::do_f() // 这个函数永远不应该被调用
{
    if( PromptUser( "pure virtual B::f called -- "
                    "abort or ignore?" ) == Abort )
        DieDieDie();
}

```

在公用的 DieDieDie() 函数中，你可以对系统进行任何操作：让程序进入调试器，或者导出堆栈数据，或者得到诊断信息。下面是一些常用方法，在大多数系统中，它们会将你带入调试器。选择你最喜欢的一个吧。

```

void DieDieDie()          // C 风格的方法：用一个 null 指针乱写
{
    // ...这是一个大家都喜欢的方法
    memset( 0, 1, 1 );
}

void DieDieDie()          // 另一个 C 风格的方法
{
    abort();
}

void DieDieDie()          // C++风格的方法：用一个 null 数据指针乱写
{
    *static_cast<char*>(0) = 0;
}

void DieDieDie()          // C++风格的方法：通过一个 null
{
    // 函数指针调用并不存在的代码
    static_cast<void(*)()>(0)();
}

void DieDieDie()          // 展开 (unwind) 至最后的“catch(...)""
{
    class LocalClass {};
    throw LocalClass();
}

void DieDieDie()          // 信奉标准者的选择方案
{
    throw std::logic_error();
}

```

```
void DieDieDie() throw() // 信奉标准者的想法，而且有很好的编译器
{
    throw 0;
}
```

明白了这一思想，就请举一反三吧。想有意摧毁一个程序，方法当然不胜枚举；但这里的目的是：让运行时期调试器（runtime debugger）尽可能地将你带到离错误最近的地方。用一个 null 指针乱写的方法总是很合算。

条款 28：受控的多态

难度：3

在 OO 模型的建立中，“Is-A” 多态是一种非常有用的工具。但有时候，对于某些类，你可能想限制某些代码多态地使用它。本条款给出了一个例子，向你展示如何得到这种效果。

请看下面的代码：

```
class Base
{
public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );
```

还有另外两个函数 f1() 和 f2()。我们的目标是：允许 f1() 在接受 Base 对象的地方多态地使用 Derived 对象，但防止其它所有函数（包括 f2()）这样做。

```
void f1()
{
    Derived d;
    SomeFunc( d ); // 允许，正确
}

void f2()
{
```

```

Derived d;
SomeFunc( d ); // 我们想防止这种情况
}

```

演示如何达到这一效果。

解答

请看下面的代码：

```

class Base
{
public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );

```

我们之所以能够在接受 `Base` 对象的地方多态地使用 `Derived` 对象，原因在于 `Derived` 从 `Base` 公有继承（这一点没什么奇怪的）。

相反，如果 `Derived` 从 `Base` 私有继承，那么，几乎没有代码可以多态地将 `Derived` 作为 `Base` 使用。之所以说“几乎”，原因在于：如果代码可以访问 `Derived` 的私有成员，它就还是可以访问 `Derived` 的私有基类，因而可以动态地将 `Derived` 替换为 `Base` 使用。正常来说，只有 `Derived` 的成员函数具有这种访问权。然而，通过 C++ 的友元特性，我们可以将类似的访问权扩充到其它外部代码中。

综合以上分析，我们得到的结果如下：

还有另外两个函数 `f1()` 和 `f2()`。我们的目标是：允许 `f1()` 在所有接受 `Base` 对象的地方多态地使用 `Derived` 对象，但防止其它所有函数（包括 `f2()`）这样做。

```

void f1()
{
    Derived d;
    SomeFunc( d ); // 允许，正确
}

```

```
void f2()
{
    Derived d;
    SomeFunc( d ); // 我们想防止这种情况
}
```

演示如何达到这一效果。

答案如下：

```
class Derived : private Base
{
public:
    void VirtFunc();
    // ...
    friend void f1();
};
```

它干净利落地解决了问题——尽管和最初版本相比，f1()获得了更大的访问权。

内存及资源管理

智能指针（smart pointer）和 std::auto_ptr 是供我们使用的重要工具，借助它们，我们可以安全高效地管理内存和其它资源——包括现实世界中的对象，如磁盘文件、数据库事务锁（database transaction lock）等。我们应该如何安全地使用 auto_ptr 呢？又该如何运用常见的设计模式来改装它、消除其常见缺陷呢？能将 auto_ptr 作为类的成员使用吗？如果你打算那样做，你知道你必须注意哪些问题吗？最后，在避免这些问题时，你能否做得更好？——即，你能否写一个专用的智能指针，从而，在一定范围内安全地保存一个成员对象？是的，你可以！当然可以……

条款 29：使用 auto_ptr

难度：5

使用 auto_ptr 时有一个常见错误，本条款就此进行了说明。问题在哪儿？如何解决？

(1) 使用 auto_ptr 时有一个常见错误，下面的函数就此进行了演示：

```
template <typename T>
void f( size_t n ) {
    auto_ptr<T> p1( new T );
    auto_ptr<T> p2( new T[n] );
    // ...更多处理 ...
    //
}
```

这段代码存在什么问题？请予以说明。

(2) 如何解决这一问题？尽可能地提供更多的方案，包括运用 Adapter 模式、替换有问题的结构、替换 auto_ptr 等。

解答

问题：不要混用数组和 auto_ptr

1. 使用 auto_ptr 时有一个常见错误，下面的函数就此进行了演示：

```
template <typename T>
void f( size_t n ) {
    auto_ptr<T> p1( new T );
    auto_ptr<T> p2( new T[n] );

    // ...更多处理 ...
    //
}
```

这段代码存在什么问题？请予以说明。

每一个 delete 必须匹配于相应形式的 new。如果使用了“单个对象”形式的 new，就必须使用“单个对象”形式的 delete；如果使用了“数组”形式的 new[], 就必须使用“数组”形式的 delete[]. 不这么做，等待你的将是不可预测的后果；在下面的例子中，那两行被稍作修改的代码就是如此：

```
T* p1 = new T;
// delete[] p1; // 错误
delete p1;      // 正确——auto_ptr 就是这样做的

T* p2 = new T[10];
delete[] p2;    // 正确
// delete p2;  // 错误——auto_ptr 就是这样做的
```

p2 的问题在于：auto_ptr 只是用来包含单个对象，所以，对于自己拥有的指针，auto_ptr 总是会调用 delete，而不是 delete[]。所以，使用普通的 delete，p1 会被正确地清除，而 p2 则不会。

如果使用了错误形式的 delete，所产生的实际后果取决于你的编译器。你能期望的最好结果是内存泄漏；更常见的后果则是内存被破坏，接踵而来的是程序崩溃。想体验一下效果吗？在你最常用的编译器上，试试下面这个完整的程序：

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

```

int c = 0;

class X {
public:
    X() : s( "1234567890" ) { ++c; }
    ~X() { --c; }
private:
    string s;
};

template <typename T>
void f( size_t n )
{
{
    auto_ptr<T> p1( new T );
    auto_ptr<T> p2( new T[n] );
}
cout << c << " "; // 报告目前存在的
} // X 对象的数目

int main()
{
    while( true )
    {
        f<X>(100);
    }
}

```

通常，这个程序要么会崩溃，要么，它会不停地输出“到目前为止已经泄漏的 X 对象的数目”。（为了增加点乐趣，在另外一个窗口中，你可以运行一个系统监视程序，用来显示系统中整个内存的使用情况。它会让你认识到，如果程序没有马上崩溃，内存泄漏会多么地有害。）

旁白：零长度数组是合法的

如果 f() 的参数为零又会怎样（例如，调用 `f<int>(0)`）？这样的话，第二个 `new` 语句就变成了 `new T[0]`，程序员一般会问：“嗯？这样正确吗？我们可以拥有一个零长度的数组吗？”

答案是：可以。零长度数组完全正确、合法而且简洁。`new T[0]` 的结果是一个指针，指向包含零个元素的数组。这个指针具有的行为特征和其它 `new [n]` 的结果一样，包括：对于这个数组，你不可能访问到 `n` 个以上的元素。对零长度数组来说，你根本就不能访问到任何元素，因为数组中没有元素。

以下摘自 C++ 标准 5.3.4 节 [expr.new] 第 7 段：

在直接 `new` 声明符 (*direct-new-declarator*) 中，如果表达式 (*expression*) 的值为零，分配函数将被调用，用来分配一个包含零个元素的数组。`new` 表达式 (*new-expression*) 返回的指针不是 `null`。[注：如果程序库分配函数被调用，返回的指针不同于指向其它任何对象的指针。]

“如果零长度的数组什么事都不能做（除了需要你记住它的地址），”你会奇怪，“那又何必允许它存在呢？”一个重要原因是：有了它，在编写动态数组分配代码时会更容易。例如上面的 f() 函数，如果在每次执行 new T[n] 调用时都需要检查参数 n 的值，代码就会变得不必要的复杂。

当然，回到主要问题：仅仅因为零长度数组合法，并不意味着我们可以让一个 auto_ptr 拥有这样一个零长度的数组，正如我们不能让 auto_ptr 拥有一个具有其它长度的数组一样。我们不能！数组删除的问题依然存在。

2. 如何解决这一问题？尽可能地提供更多的方案，包括运用 Adapter 模式、替换有问题的结构、替换 auto_ptr 等。

可供选择的方案有不少（一些较好，一些较差）。这里列出了四个：

方案 1：打造自己的 auto_array

这一方案可以比听上去更容易，也可以更难。

方案 1(a)：...通过从 auto_ptr 派生（得分：0/10）

坏主意！例如，你必须重新实现所有的拥有权（ownership）及辅助类（helper class）语义（译注：有关 auto_ptr 的拥有权及辅助类语义，可以参阅[Stroustrup00]14.2.2 节），因而得在大量的重复工作中疲于奔命。这种方案只可能由真正的专家去尝试，但真正的专家永远不会去尝试，因为有更容易的方法。

优点：几乎没有。

缺点：举不胜举。

方案 1(b)：...通过复制 auto_ptr 的代码（得分：8/10）

这个主意是，取出程序库实现中的 auto_ptr 代码，复制它（将其更名为 auto_array 或其它什么名字），然后将其中的 delete 语句改为 delete[] 语句。

优点：

(a) *易于实现（只需一次）*。我们无需手工编写自己的 auto_array，但同时，auto_ptr 的所有语义自动地得以保留；将来维护这段代码的程序员只要熟悉 auto_ptr，就不会对这里的代码大惊小怪。

(b) *没有空间或时间上的显著开销*。

缺点：

难以维护。当你升级到新的编译器/程序库版本时，或是更换编译器/程序库供货商时，你得十分小心地使 auto_array 与之同步。

方案 2：运用 Adapter 模式（得分：7/10）

在我的一次演讲之后，我和来自 C++ World 的一位佳宾 Henrik Nordberg 曾就这一问题进行过讨论，这个方案就来自于那次讨论。对那段有问题的代码，Henrik 的第一反应是：最简单的办法应该是写一个适配器（adapter）来让标准 auto_ptr 工作正常，而不是重写 auto_ptr，或使用其它什么方法。这个思路尽管还是有那么一点点缺陷，但的确具有它的优点，值得在这里探讨。

以下就是它的设计思想。我们不这么写：

```
auto_ptr<T> p2( new T[n] );
```

而是这么写：

```
auto_ptr< ArrDelAdapter<T> >
p2( new ArrDelAdapter<T>(new T[n]) );
```

其中，ArrDelAdapter（即 Array Deletion Adapter）具有一个“参数为 T* 指针的构造函数”；在析构函数中，它用那个指针调用 delete[]：

```
template <typename T>
class ArrDelAdapter {
public:
    ArrDelAdapter( T* p ) : p_(p) { }
    ~ArrDelAdapter() { delete[] p_; }
    // "->"、"T*" 等运算符以及其它辅助函数
private:
    T* p_;
};
```

既然只有一个 ArrDelAdapter<T> 对象，~auto_ptr() 中“单个对象”形式的 delete 语句就不会有问题。因为~ArrDelAdapter<T> 对数组正确地调用了 delete[]，最初的问题已经得以解决。

的确，这也许不是世界上最优雅最漂亮的手法，但至少我们无须手工编写自己的 auto_array 模板！

优点：

(a) 易于实现（利用了固有的功能）。我们无须编写 auto_array。实际上，我们还自动保留了 auto_ptr 的所有语义；将来维护这段代码的程序员只要用惯了 auto_ptr，就不

会对此大惊小怪。

缺点：

(a) 可读性差。代码过于啰唆。

(b) (可能) 难以使用。在 f 中，后面的代码只要使用了 p2 的值，那些代码就得做句法上的改变；额外的间接性往往带来更多的麻烦。

(c) 带来空间和时间上的开销。这个方案需要额外的空间，因为它需要存储每一个数组所需的 Adapter 对象。它还需要额外的时间，因为它需要执行多达两次的内存分配（这一点可以通过重载 new 运算符得以改善），这样一来，调用者代码每次访问数组时，都多了一层间接性。

说了这么多，再补充一句：即使在当前这一特定场合，可能还会有其它更好的方案，但看到大家能够立即想到使用 Adapter 模式，我还是很高兴。Adapter 是广泛应用的核心模式之一，每一位程序员都应该了解它。

设计准则

了解并使用设计模式。

关于方案 2，最后要说的一点是，下面这样写：

```
auto_ptr< ArrDelAdapter<T> >
p2( new ArrDelAdapter<T>(new T[n]) );
```

和这样写没什么大的不同：

```
auto_ptr< vector<T> > p2( new vector<T>(n) );
```

就此思考一会儿。例如，问问自己：“和单纯写一句 vector p2(n); 相比，动态地分配这个 vector 会有什么好处吗？如果有，好处在哪儿？”答案请参见方案 4。

方案 3：用手工编写的异常处理逻辑取代 auto_ptr (得分：1/10)

函数 f()之所以要使用 auto_ptr，是为了获得对象自动清除的好处，还可能是为了得到异常安全。既然如此，我们可以为 p2 数组剥掉 auto_ptr 这层外衣，手工编写自己的异常处理逻辑。

即，我们不这么写：

```
auto_ptr<T> p2( new T[n] );
//
// ... 更多处理 ...
//
```

取而代之的是，我们这么写：

```
T* p2( new T[n] );
try {
    //
    // ... 更多处理 ...
    //
}
delete[] p2;
```

优点：

- (a) 易于使用。对于那段使用了 p2 的“更多处理”代码块，这个方案几乎不会给它带来影响；需要做的可能只是删除每一个出现的 get() 而已。
- (b) 没有空间或时间上的显著开销。

缺点：

- (a) 难以实现。这个方案可能涉及大量的代码改动，远远多于上面所提到的工作量。因为，对于新的 T 对象，无论函数如何退出，p1 的 auto_ptr 都会自动清除它；然而，要清除 p2，针对函数退出时可能经过的每一条路径，我们都必须提供清除代码。例如，考虑这样的情形：“更多处理”代码块中包含更多的分支，其中一些分支以“return”结束……
- (b) 缺乏健壮性。参见(a)：在每一个执行路径上，我们都提供了合适的清除代码吗？
- (c) 可读性差。参见(a)：额外的“清除操作”逻辑很可能掩盖函数的正常逻辑。

方案 4：用 vector 取代数组（得分：9.5/10）

在前面的方案中，我们遇到的大部分问题应该归因于：我们使用的是 C 风格的数组。一个事实是：只要合适，用 vector 取代 C 风格的数组会更好，况且，这样做几乎总是合适的。毕竟，vector 之所以存在于标准库之中，主要原因就在于，它可以替代 C 风格的数组，而且提供了更高的安全性，更便于使用！

所以，我们不这样写：

```
auto_ptr<T> p2( new T[n] );
```

取而代之的是：

```
vector<T> p2( n );
```

优点：

- (a) 易于实现（使用固有的功能）。我们还是无须编写 auto_array。

(b) 可读性好。熟悉标准容器的人（每个人现在都应该这样！）会立刻明白是怎么回事。

(c) 程序健壮性提高。既然将内存管理的细节丢给了底层，我们的代码（通常）将更为简化。我们无须管理 T 对象的内存缓冲区——那是 `vector<T>` 对象的工作。

(d) 没有空间和时间上的显著开销。

缺点：

(a) 语法改变。在 f 中，后面的代码只要使用了 p2 的值，就需要作语法上的改变，尽管这种改变十分简单、不像方案 2 要求得那样强烈。

(b) (有时) 使用性改变。如果 T 类型不支持拷贝构造和赋值，任何标准容器（包括 `vector`）就都不能通过这样的 T 类型实例化。大多数类型都支持拷贝构造和赋值。但如果某种类型不支持，这个方案就对它不适用。

注意，和传递或返回 `auto_ptr` 相比，通过值（`value`）传递或返回 `vector` 所带来的开销更大。但我认为，这一异议和我们的话题有点无关，因为这是一种不公平的对比。如果想得到同样的效果，你可以传递 `vector` 的指针或引用。

设计准则

尽量使用 `vector`，不要使用内建的（C 风格的）指针。

条款 30：智能指针成员，之一：

`auto_ptr` 存在的问题

难度：5

大多数 C++ 程序员都知道，如果一个类中有指针成员，我们就得特别小心。但如果一个类中有 `auto_ptr` 成员呢？为了使自己以及用户的处境更加安全，我们能否针对类的成员，专门设计一个智能指针类呢？

(1) 请看下面这个类：

```
// 例 30-1
//
class X1
{
    // ...
}
```

```
private:
    Y* y_;
};
```

如果 X1 对象拥有 (owns) 所指向的 Y 对象，那么，X1 的设计者为什么不能使用编译器自动生成的析构函数、拷贝构造函数和拷贝赋值函数？

(2) 下面的做法有什么优点和缺点？

```
// 例 30-2
//
class X2
{
    // ...
private:
    auto_ptr<Y> y_;
};
```

解答

回顾：指针成员存在的问题

1. 请看下面这个类：

```
// 例 30-1
//
class X1
{
    // ...
private:
    Y* y_;
};
```

如果 X1 对象拥有 (owns) 所指向的 Y 对象，那么，X1 的设计者为什么不能使用编译器自动生成的析构函数、拷贝构造函数和拷贝赋值函数？

简单地说，如果 X1 拥有所指向的 Y，编译器生成的上述函数不会提供正确的功能。首先，如果 X1 拥有 Y，X1 就得有某个函数（可能是 X1 的构造函数）来创建这个 Y 对象，还得有另外一个函数（可能是析构函数 X1::~X1()）来删除它：

```
// 例 30-1(a): 拥有权 (ownership) 语义
//
{
    X1 a; // 分配一个新的 Y 对象并指向它
    // ...
```

```

} // 当 a 走出生存空间并被摧毁
// 时，它会删除所指向的 Y

```

那么，如果使用“以成员为单位（memberwise）”的缺省拷贝构造，就会导致多个 X1 对象指向同一个 Y 对象，这将会带来奇怪的后果。例如，修改一个 X1 对象会同时改动另一个对象的状态；另外，`delete` 操作会发生两次：

```

// 例 30-1(b)：共享，以及两次 delete
//
{
    X1 a;    // 分配一个新的 Y 对象并指向它

    X1 b( a ); // b 现在和 a 一样，指向同一个 Y 对象

    // ... 操作 a 和 b 会修改
    // 同一个 Y 对象 ...

} // 当 b 走出生存空间并被摧毁时，它会
// 删除所指向的 Y...a 也会这样做。噢！

```

如果使用“以成员为单位”的缺省拷贝赋值，也会导致多个 X1 对象指向同一个 Y 对象，从而，同样会产生状态共享和两次 `delete` 的问题；更有甚者，当一些对象永远没有被删除时，它还会造成资源泄漏：

```

// 例 30-1(c)：共享、两次 delete，再加上资源泄漏
//
{
    X1 a;    // 分配一个新的 Y 对象并指向它

    X1 b;    // 分配一个新的 Y 对象并指向它

    b = a;    // b 现在和 a 一样，指向同一个 Y 对象，
    // 但没有任何对象指向 b 创建的 Y 对象

    // ... 操作 a 和 b 会修改
    // 同一个 Y 对象 ...

} // 当 b 走出生存空间并被摧毁时，它会
// 删除所指向的 Y...a 也会这样做。噢！

// b 分配的 Y 对象永远没有被删除

```

在其它场合，我们一般会遵循好的习惯，将普通指针包装在管理者对象中，使之拥有指针并简化清除工作。如果 Y 成员也由这样一个管理者对象而不是一个普通指针来保存，情况会有所改善吗？这给我们带来下一个重要的问题。

关于 auto_ptr 成员

2. 下面的做法有什么优点和缺点？

```
// 例 30-2
//
class X2
{
    // ...
private:
    auto_ptr<Y> y_;
};
```

这会带来一定的好处，但没有完全解决问题，自动生成的拷贝构造和拷贝赋值函数还是会做错误的事。只不过，是在做不同的错事。

首先，如果 X2 具有用户自定义的构造函数，让这些构造函数具有异常安全性会更容易，因为，如果构造函数抛出异常，auto_ptr 会自动执行清除工作。但是，在 auto_ptr 对象获得 Y 对象的拥有权之前，X2 的设计者还是得分配自己的 Y 对象并通过一个普通指针来保存它——无论这一过程多么短暂。

第二，现在，编译器自动生成的析构函数的确会做正确的事。当 X2 对象走出生存空间并被摧毁时，auto_ptr<Y> 的析构函数会删除它拥有的 Y 对象，自动执行清除工作。即便如此，这里还是有一个很微妙的限制，它曾经引起过我的注意：如果你必须依赖于自动生成的析构函数，那么，在使用了 X2 的每一个编译单元（translation unit）中，这个析构函数都得提供定义。这几乎意味着，对使用了 X2 的任何人，Y 的定义都必须可见。举个例子，如果 Y 是一个 Pimpl，那就不是件好事，因为 Pimpl 的全部意义在于对 X2 的用户隐藏 Y 的定义。所以，你可以依赖自动生成的析构函数，但只有将 Y 的完整定义提供给 X2 才行（例如，x2.h 包含 y.h）：

```
// 例 30-2(a): Y 必须被定义
//
{
    X2 a; // 分配一个新的 Y 对象并指向它
    // ...
} // 当 a 走出生存空间并被摧毁时，它删除
// 所指向的 Y；只有存在 Y 的完整定义时，
// 这才会发生。
```

如果你不想提供 Y 的定义，就必须显式地写出 X2 的析构函数，即使函数体为空。

第三，至于自动生成的拷贝构造函数，它不再具有例 30-1(b)所描述的两次 delete 问题。这是个好消息。不太好的消息是，这个自动生成的版本带来了另一个问题——

数据偷窃 (grand theft)。“被创建的 X2 对象”会偷走“被拷贝的 X2 对象”中的 Y 对象，包括 Y 对象的所有信息。

```
// 例 30-2(b): 数据偷窃指针
//
{
    X2 a; // 分配一个新的 Y 对象并指向它

    X2 b( a ); // b 偷走了 a 中的 Y 对象，使得 a 的
                // y_成员成为一个 null auto_ptr

    // 如果试图使用 a 的 y_成员，操作将会出错；如果
    // 幸运的话，这一问题将会立即表现为一个程序崩
    // 溃；否则，它可能会表现为一个难以诊断的间歇
    // 性错误
}
```

数据偷窃问题中仅有的一点安慰（虽然太少）是：如果盗窃似的行为会发生，自动生成的 X2 的拷贝构造函数至少会提供某种合理的警告。为什么这样说呢？因为，这个函数的原型将是 X2::X2(X2&)。请注意，它的参数是一个指向非 const 对象的引用。^⑩毕竟，auto_ptr 的拷贝构造函数就是这样做的，所以 X2 自动生成的拷贝构造函数也必须这么做。这确实是细微之处，但它至少可以防止对 const X2 进行拷贝。

最后，说说自动生成的拷贝赋值运算符。它不再具有例 30-1(c) 所描述的两次 delete 和资源泄漏问题。这是个好消息。唉，同样的，还是有一些不太好的消息，数据偷窃问题同样存在：赋值过程中的“目标 X2 对象”会偷走“源 X2 对象”中的 Y 对象，包括 Y 对象的所有信息；除此以外，它还会（可能过早地）删除掉自己最初拥有的 Y 对象。

```
// 例 30-2(c): 更厉害的数据偷窃指针
//
{
    X2 a; // 分配一个新的 Y 对象并指向它

    X2 b; // 分配一个新的 Y 对象并指向它

    b = a; // b 删除了自己的 Y，偷走了 a 的 Y,
            // 使得 a 拥有一个 null auto_ptr

    // 和例 30-2(b)一样，任何试图通过 a 使用其
    // y_成员的操作将是灾难性的
}
```

^⑩ 可惜，参数的“非 const 性”是不可见的。函数由编译器无声无息地生成，你永远不能在人眼所能看到的代码中读到这些函数的原型。

和前面类似，至少这种偷窃行为会被提示，因为自动生成的函数会被声明为 `X2& X2::operator=(X2&)`，这就相当于广告天下（尽管是用极小的字体，而不是头版头条标题②）：操作数可以被修改。

总而言之，`auto_ptr` 确实带来了某些好处，特别是，它为构造函数和析构函数自动完成了清除工作。然而，在本例中，它没有通过自身回答最初的主要问题——我们必须为 `X2` 提供自己的拷贝构造和拷贝赋值函数，或者，如果拷贝操作对这个类没有意义，我们必须禁止它们。关于这一点，我们可以采用更专用一点的方式，从而做得更好。

ValuePtr 主题变奏

本短系列的正题是逐步求精地设计出一个 `ValuePtr` 模板，对于以上介绍的各种应用场合，这个 `ValuePtr` 将比 `auto_ptr` 更合适。

就异常规范说明一点：异常规范没有你想象的那么有用；至于原因，我不想在此深入讨论。真正重要的是，对于一个函数，你应该知道它会抛出什么异常；如果一个函数给出了“不抛出异常（`nothrow`）”的保证，宣告它绝对不会抛出异常，你就更得清楚地知道。你不需要用异常规范来证明你的行为，所以我作如下声明：

在这两个条款演示的每一个 `ValuePtr<T>` 版本中，所有成员函数都提供“不抛出异常”的保证；一种情况除外：在以 `ValuePtr<U>`（其中 `U` 可以为 `T`）为源对象执行构造或赋值时，有可能会抛出“来自 `T` 的构造函数的异常”。

下面切入正题，请看条款 31。

条款 31：智能指针成员，之二：

设计 `ValuePtr`

难度：6

为了使自己以及用户的处境更加安全，我们能否将内部的成员专门设计一个智能指针类？

(1) 写一个合适的 `ValuePtr` 模板，它可以像下面这样使用：

```
// 例 31-1
//
class X
{
    // ...
private:
    ValuePtr<Y> y_;
};
```

并分别适应以下三种特定场合：

- (a) 不允许对 ValuePtr 进行拷贝和赋值。
- (b) 允许对 ValuePtr 进行拷贝和赋值，且具有这样的语义：在创建其 Y 对象的拷贝时，使用的是 Y 的拷贝构造函数。
- (c) 允许对 ValuePtr 进行拷贝和赋值，且具有这样的语义：如果 Y 提供了一个 virtual Y::Clone() 函数，在创建 Y 对象的拷贝时，将使用这个函数；否则，如果 Y 没有提供这样一个函数，将使用 Y 的拷贝构造函数。

解答

一个简单的 ValuePtr：绝对的拥有权

1. 写一个合适的 ValuePtr 模板，它可以像下面这样使用：

```
// 例 31-1
//
class X
{
    // ...
private:
    ValuePtr<Y> y_;
};
```

我们将考虑三种情况。在所有三种情况中，构造函数的好处还是得保证：清除工作自动进行、X::X() 的设计者只需做少量工作就可以保证异常安全、避免构造函数失败造成资源泄漏。^⑩ 同样，在所有三种情况中，对于析构函数的限制依然存在：要么，Y 的完整定义必须和 X 如影相随；要么，必须显式地提供 X 的析构函数，即使函数体为空。

并分别适应以下三种特定场合：

- a) 不允许对 ValuePtr 进行拷贝和赋值。

这实在没什么好做的：

```
// 例 31-2 (a): 简单情况:
// 没有拷贝和赋值的 ValuePtr
//
template<typename T>
```

^⑩ 还有一种可能，即，让 ValuePtr 来对它所拥有的 Y 对象进行构造；但为了清晰性，我将忽略这种情况。因为，如果那样做的话，ValuePtr<Y> 几乎就和“Y 的值”具有相同的语义，那么拜托回答这样一个问题：为什么不干脆使用普通的旧式 Y 成员呢？

```

class ValuePtr
{
public:
    explicit ValuePtr( T* p = 0 ) : p_( p ) { }

    ~ValuePtr() { delete p_; }

```

当然，还得有什么途径来访问指针；所以，就像 `std::auto_ptr` 那样，得提供下面这样的东西：

```

T& operator*() const { return *p_; }

T* operator->() const { return p_; }

```

还需要什么？`auto_ptr` 提供了 `reset()` 和 `release()` 两个函数，对很多智能指针类型来说，提供类似这两个函数的功能将很有意义；因为这样一来，对于一个 `ValuePtr` 拥有哪一个对象，用户就可以任意改变。初看上去，加入这些功能似乎是个好主意，因为它们符合 `ValuePtr` 的设计目的和使用方向——即，将 `ValuePtr` 作为一个类成员来使用。那么，请考虑条款 22 中的例 22-2 和例 22-3。在那两个例子中，类成员拥有的是一个 `Pimpl` 指针，其目的是想撰写一个具有异常安全性的赋值运算符。于是，你就需要一种方法来交换 `ValuePtr`，但不能拷贝所拥有的对象。但是，提供类似 `reset()` 和 `release()` 那样的函数不是达到这一目的的正确方法，因为它允许了太多的操作。是的，为了执行交换操作或获得异常安全，它可以让用户去做需要做的事情；但同时，它也为其它许多（不必要的）操作敞开了大门，但那些操作也许有违 `ValuePtr` 的设计目的，而且，如果滥用那些操作还会造成问题。

那么，该怎么做？不要提供过于全面的功能；相反，好好理解你的需求，只提供真正需要的功能：

```

void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }

private:
    T* p_;

    // 禁止拷贝
    ValuePtr( const ValuePtr& );
    ValuePtr& operator=( const ValuePtr& );
};


```

我们获得指针的拥有权，后来删除之。我们处理空指针的情况；而且，拷贝和赋值被特别禁止，做法很平常——将它们声明为私有成员但不提供定义。将构造函数声明为 `explicit` 是一种好的做法，它避免了隐式转换；对于这种转换，`ValuePtr` 的用户绝对不会需要。

这一部分非常浅显，但下面几步蕴涵着某些微妙之处。

拷贝构造和拷贝赋值

b) 允许对 ValuePtr 进行拷贝和赋值，且具有这样的语义：在创建其 Y 对象的拷贝时，使用的是 Y 的拷贝构造函数。

下面的方法满足这个要求，但没有做到应有的通用性。它和例 31-2(a)相同，但为拷贝构造和拷贝赋值提供了定义：

```
// 例 31-2(b): 允许拷贝和赋值
// 的 ValuePtr, 第 1 步
//
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr( T* p = 0 ) : p_( p ) { }

    ~ValuePtr() { delete p_; }

    T& operator*() const { return *p_; }

    T* operator->() const { return p_; }

    void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }

    //--- 新增代码开始 -----
    ValuePtr( const ValuePtr& other )
        : p_( other.p_ ? new T( *other.p_ ) : 0 ) { }
    //--- 新增代码结束 -----
```

注意，检查 other 的指针是否为空很重要。但 operator=() 是通过拷贝构造来实现的，所以我们只需将检查工作放在一个地方。

```
ValuePtr& operator=( const ValuePtr& other )
{
    ValuePtr temp( other );
    Swap( temp );
    return *this;
}
//--- 新增代码结束 -----
```

```
private:
    T* p_;
};
```

这满足前面提出的要求，因为在预定的使用场合下，我们只需要从“管理 T 类型的 ValuePtr” 拷贝或赋值，不需要从“管理任何其它类型的 ValuePtr” 拷贝或赋值。如果确信这是我们的需求，上面这个 ValuePtr 已经很不错了。但无论何时，当我们在设计一个类时，如果不需要很多额外的工作就可以提供新的功能，而且这些新功能在将来会对用户更有用处，那么，我们至少应该考虑到设计的可扩充性。与此同时，在“提

高设计的可复用性”和“避免设计过度（overengineering）的危险”之间，我们需要做出权衡：所谓设计过度，指的是对一个简单的问题提供过于复杂的方案。这给我们带来下一个问题。

模板化的构造和模板化的赋值

考虑这样一个问题：如果我们允许“将来可以在不同类型的 ValuePtr 之间进行拷贝和赋值”，这会对例 31-2(b)的代码带来哪些影响？也就是说，我们希望：如果 X 可以转换为 Y，就可以将 ValuePtr<X>拷贝或赋值给 ValuePtr<Y>。

答案是，这种影响可以做到最小（minimal）。我们可以复制（duplicate）出拷贝构造函数和拷贝赋值函数的模板化版本，这只要在它们前面加上 `template<typename U>`，并且，取一个类型为 `ValuePtr<U>&` 的参数，如下所示：

```
// 例 31-2(c): 允许拷贝和赋值
// 的 ValuePtr, 第 2 步
//
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr( T* p = 0 ) : p_( p ) { }

    ~ValuePtr() { delete p_; }

    T& operator*() const { return *p_; }

    T* operator->() const { return p_; }

    void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }

    ValuePtr( const ValuePtr& other )
        : p_( other.p_ ? new T( *other.p_ ) : 0 ) { }

    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

    //--- 新增代码开始 -----
    template<typename U>
    ValuePtr( const ValuePtr<U>& other )
        : p_( other.p_ ? new T( *other.p_ ) : 0 ) { }

    template<typename U>
    ValuePtr& operator=( const ValuePtr<U>& other )
    {
        ValuePtr temp( other );
```

```

        Swap( temp );
        return *this;
    }

private:
    template<typename U> friend class ValuePtr;
    //--- 新增代码结束 -----
    T* p_;
};

```

注意到我们躲开了一个陷阱吗？——我们还是得写出非模板形式的拷贝和赋值函数。这样做是为了禁止自动生成的版本；因为，模板化的构造函数绝不是拷贝构造函数，模板化的赋值运算符也绝不是拷贝赋值运算符。关于这一点的更多介绍，请参见 [Exceptional C++ \[Sutter00\] 条款 5](#)。

还有一个微妙之处要当心，但幸亏不是个大问题。我甚至可以说，那不是 ValuePtr 的设计者的责任。这个微妙之处是：无论是模板形式或非模板形式的拷贝和赋值函数，在我们执行切割 (slicing) 操作的情况下，源对象 other 还是可以拥有一个指向派生类型的指针。例如：

```

class A {};
class B : public A {};
class C : public B {};

ValuePtr<A> a1( new B );
ValuePtr<B> b1( new C );

// 调用拷贝构造函数，切割
ValuePtr<A> a2( a1 );

// 调用模板化的构造函数，切割
ValuePtr<A> a3( b1 );

// 调用拷贝赋值函数，切割
a2 = a1;

// 调用模板化的赋值函数，切割
a3 = b1;

```

我之所以指出这一点，原因在于，我们不应该忘记把这类事情写在 ValuePtr 的说明文档中，而且最好放在“不要这样做”一节中，从而提醒用户注意。作为 ValuePtr 的设计者，我们没有其它办法能够在代码中阻止这类误用。

那么，哪一个方案是问题 1-(b) 的正确答案呢？例 31-2(b) 还是例 31-2(c)？二者都是不错的方案，这的确需要你的判断和经验，它取决于你在“提高设计的复用性”和“避免设计过度”之间如何作出权衡。可以想见，“最小化设计”的拥护者会自然而然地使用方案 31-2(b)，因为它完全满足最小需求。我还可以想象，如果 ValuePtr 是在一个库

中，它由一个团队写成，并被几个不同的开发小组共享；这种情况下，因为方案 31-2(c)具有可复用性并可以避免重复开发，使用方案 31-2(c)就会节省整个开发的工作量。

通过 traits 增加可扩充性

但如果 Y 有一个 Clone() 虚拟函数又该怎么办？从条款 30 的例 30-1 看来，X 似乎总会创建自己的 Y 对象，但实际上，这个对象有可能是从一个工厂（factory）（译注：一个封装了一组对象创建操作的抽象类，参见[Stroustrup00]第 12.4.4 节）或从某个派生类型的 new 表达式中得到的。可见，在这种情况下，所拥有的 Y 对象可能根本就不是一个真正的 Y 对象，而是从 Y 派生的某种类型的对象，那么，如果将它作为一个 Y 来拷贝，最好的结果是会切割这个对象，最坏的结果是致使这个对象不可用。在这种情况下，一个常用的技术是为 Y 提供一个专门的 Clone() 虚拟成员函数；这样一来，即使不知道所指向对象的完整类型，完整的拷贝操作也可以执行。

如果有人想用 ValuePtr 来保存这样一种对象，这个对象就只能用其它某个函数来拷贝，而不是拷贝构造函数，这种情况下该怎么办？这是我们最后一个要点。

c) 允许对 ValuePtr 进行拷贝和赋值，且具有这样的语义：如果 Y 提供了一个 virtual Y::Clone() 函数，在创建 Y 对象的拷贝时，将使用这个函数；否则，如果 Y 没有提供这样一个函数，将使用 Y 的拷贝构造函数。

在 ValuePtr 模板中，我们不知道 T 的真正类型是什么，我们不知道它是否有一个 Clone() 虚函数。因此，我们不知道如何通过正确的方式拷贝它。是这样的吗？

一个方案是使用 traits，C++ 标准库本身就广泛运用了这一技术。（关于 traits 的更多介绍，请参见条款 4）要实现一个基于 traits 的方案，让我们先对例 31-2(c) 做少量的修改，删除一些冗余代码。你会注意到，模板化的构造函数和拷贝构造函数都必须检查源对象是否为 null。我们可以将所有这些工作放在一个单独的地方，并用一个单独的 CreateFrom() 函数来构造新的 T 对象。一会儿，我们还会看到这样做的另一个理由。

```
// 例 31-2(d)：允许拷贝和赋值
// 的 ValuePtr，对例 31-2(c)
// 做了少量的改动
//
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr( T* p = 0 ) : p_( p ) { }

    ~ValuePtr() { delete p_; }
```

```

T& operator*() const { return *p_; }

T* operator->() const { return p_; }

void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }

ValuePtr( const ValuePtr& other )
: p_( CreateFrom( other.p_ ) ) {} // 作了改动

ValuePtr& operator=( const ValuePtr& other )
{
    ValuePtr temp( other );
    Swap( temp );
    return *this;
}

template<typename U>
ValuePtr( const ValuePtr<U>& other )
: p_( CreateFrom( other.p_ ) ) {} // 作了改动

template<typename U>
ValuePtr& operator=( const ValuePtr<U>& other )
{
    ValuePtr temp( other );
    Swap( temp );
    return *this;
}

private:
    //--- 新增代码开始 -----
    template<typename U>
    T* CreateFrom( const U* p ) const
    {
        return p ? new T( *p ) : 0;
    }
    //--- 新增代码结束 -----

    template<typename U> friend class ValuePtr;

    T* p_;
};

```

现在，“以不同方式对 T 进行拷贝”的所有信息可以被封装起来，放在 CreateFrom() 这个好地方。

运用 traits

现在，我们可以通过下面的方式运用 traits 技术。请注意，这不是运用 traits 的唯一方式。还请注意，要通过不同的方式拷贝 T，除了运用 traits 之外也还有其它的方式。我决定使用一个 traits 类模板，它有一个静态 Clone() 成员函数，这个函数可以调用任何所需操作，以完成实际的复制（cloning）工作；因而，我们可以将其看作是一个适配器

(adapter)。这种做法遵循 `char_traits` 的风格：例如，有了 `char_traits`, `basic_string` 就可以将“字符处理策略的定义工作”委托给一个 `traits` 类。（作为一种选择，`traits` 类也可以提供 `typedef` 或其它辅助手段，这样，`ValuePtr` 模板就可以知道要做什么，但同时，这些工作还是得由它自己来做。我不喜欢这种做法，因为这似乎是一种不必要的责任分工。对于这样一种简单的事情，何苦在一个地方判断出应该做什么事，然后又在另一个地方完成实际工作呢？）

```
template<typename T>
class ValuePtr
{
    // ...

    template<typename U>
    T* CreateFrom( const U* p ) const
    {
        // 决定“做正确的事”... 但如何做？
        return p ? VPTraits<U>::Clone( p ) : 0;
    }
};
```

我们希望 `VPTraits` 是一个完成实际复制 (cloning) 工作的模板，其中，主模板 (main template) 中 `Clone()` 的实现使用的是 `U` 的拷贝构造函数。有两点要注意：首先，既然 `ValuePtr` 负责对空指针进行检查，`VPTraits::Clone()` 就不必这么做。第二，为了在 `T` 为基类、`U` 为派生类的情况下可以正确地处理多态操作，那么，在 `U` 和 `T` 不同的情况下，只有 `U*` 可以转换为 `T*`，这个函数才可以通过编译。

```
template<typename T>
class VPTraits
{
public:
    static T* Clone( const T* p ) { return new T( *p ); }
};
```

这样，针对任何一个不想使用拷贝构造的类型 `Y`，`VPTraits` 可以像下面那样被特殊化。例如，假设某个 `Y` 有一个 `Y* CloneMe()` 虚函数，某个 `Z` 有一个 `void CopyTo(Z&)` 虚函数。那么，我们可以将 `VPTraits<Y>` 特殊化，让那些函数完成复制工作：

```
// 用户必须做的绝大部分工作就在这里，
// 而且只需要在一个地方做一次
//
template<>
class VPTraits<Y>
{
public:
    static Y* Clone( const Y* p )
    { return p->CloneMe(); }
```

```

};

template<>
class VPTraits<Z>
{
public:
    static Z* Clone( const Z* p )
    { Z* z = new Z; p->CopyTo(*z); return z; }
};

```

这个方案更加出色，因为它大大避免了设计修改，同时提高了设计的可扩充性；它适用于具有任何风格和原型的 `CloneMe()` 函数、甚至将来的某个函数。在底层，`Clone()` 可以通过任何合适的方式创建对象，唯一可见的结果是一个指向新对象的指针——这是提高封装性的又一个好论据。

若干个世纪之后，`ValuePtr` 的设计者早已化为了尘土，一个崭新的类型 `Y` 诞生了；如果想通过那个崭新的 `Y` 使用这个古老的 `ValuePtr`，生活在第四个新千年的 `Y` 的用户（或设计者）只需要将 `VPTraits` 特殊化一次，然后，在她的代码中的任何地方，她都可以使用 `ValuePtr<Y>`。那的确太容易了。如果 `Y` 没有 `Clone()` 虚函数，`Y` 的用户（或设计者）甚至连那些工作都不用做，也就是说，不用做任何工作也可以使用 `ValuePtr`。

简短的结尾：既然 `VPTraits` 仅有一个静态函数模板，为什么我们要写一个类模板而不是一个函数模板呢？——主要目的在于封装性（尤其是，更好地管理名称）和可扩充性。我们不希望用自由函数搅乱全局名字空间。有一种做法是：无论 `ValuePtr` 本身在哪个名字空间，将函数模板也放在这一名字空间范围之中。但这样一来，和同一名字空间中的其它代码相比，函数模板和 `ValuePtr` 的耦合性会变得更加紧密。在目前看来，`Clone()` 函数模板或许是我们需要的唯一一种 trait，但如果明天我们需要一个新的 trait 怎么办？如果新增的 traits 是函数，我们就不得不继续用自由函数将名字空间搅得一团糟。但如果新增的 traits 是 `typedef` 甚至是类呢？——`VPTraits` 是一个把这一切封装起来的好地方。

运用了 Traits 的 ValuePtr

下面是新的 `ValuePtr` 代码，它运用了具有复制功能的 traits，并满足前面问题 c) 中的所有要求。^⑤ 注意，和例 31-2(d) 相比，这里只对 `ValuePtr` 改动了一行代码，并提供了

^⑤ 有可能，一些人会提供一个 traits 对象，作为 `ValuePtr` 模板的另一个参数 `Traits`，这个参数的缺省值为 `VPTraits<T>`，就像 `std::basic_string` 所做的那样。

```

template<typename T, typename Traits = VPTraits<T> >
class ValuePtr { /*...*/ };

```

这样一来，在同一个程序拷贝中，用户甚至可能通过不同的方式拥有不同的 `ValuePtr<X>` 对象。在我们这个特定场合下，这种额外的灵活性似乎没有多大的意义，所以我不准备这样做——但要知道有这种可能。

一个新模板，这个模板仅有一个简单的函数。这就是我所说的最小（minimal）影响，它换来的是前面说过的所有灵活性。

```
// 例 31-2(e)：允许拷贝和赋值的
// ValuePtr，提供了基于 traits
// 的完整定制功能
//
//---- 新增代码开始 -----
template<typename T>
class VPTraits
{
    static T* Clone( const T* p ) { return new T( *p ); }
};

//---- 新增代码结束 -----


template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr( T* p = 0 ) : p_( p ) { }

    ~ValuePtr() { delete p_; }

    T& operator*() const { return *p_; }

    T* operator->() const { return p_; }

    void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }

    ValuePtr( const ValuePtr& other )
        : p_( CreateFrom( other.p_ ) ) { }

    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

    template<typename U>
    ValuePtr( const ValuePtr<U>& other )
        : p_( CreateFrom( other.p_ ) ) { }

    template<typename U>
    ValuePtr& operator=( const ValuePtr<U>& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

private:
    template<typename U>
```

```

T* CreateFrom( const U* p ) const
{
//--- 新增代码开始 -----
    return p ? VPTraits<U>::Clone( p ) : 0;
//--- 新增代码结束 -----
}

template<typename U> friend class ValuePtr;

T* p_;
};

```

应用实例

下面的例子是一个类，它使用了 ValuePtr 的最终版本；在这个类中，我们演示了其主要函数（构造、析构、拷贝、赋值）的典型实现，但省略了细节性问题——譬如，根据 Y 提供了定义或未提供定义，析构函数是否也应该提供（或内联），等等。

```

// 例 31-3: ValuePtr 应用实例
//
class X
{
public:
    X() : y_( new Y(/*...*/) ) { }

    ~X() { }

    X( const X& other ) : y_( new Y(*other.y_) ) { }

    void Swap( X& other ) { y_.Swap( other.y_ ); }

    X& operator=( const X& other )
    {
        X temp( other );
        Swap( temp );
        return *this;
    }

private:
    ValuePtr<Y> y_;
};

```

总结

我希望，你在本条款能够吸取这样一条经验：在设计的时候，可扩充性一定要牢记在心。

设计准则

一般情况下，尽量提高设计的通用性。

在避免“设计过度”的陷阱时，总要保持长远的眼光。你可以每次否决它，但总要清楚它的存在。这样，从长远来看，你和你的用户就会节省大量的时间和人力；在长远的将来，你的用户在使用他们自己编写的新类时，也会乐于复用你的代码。

自由函数与宏

在本书的其它章节，我们集中讨论了 C++ 对泛型程序设计和面向对象程序设计的强大支持。不过，对于早已熟悉的普通函数、甚至从 C 语言继承而来（可以这么说）的宏，还是有某些值得探讨之处。

你使用过支持嵌套函数的程序设计语言吗？如果使用过，你或许想知道，是否有可能（或值得）在 C++ 中得到同样的效果；关于这一点，你可以在条款 33 中找到答案。在大多数场合，预处理宏是彻头彻尾的肇事者，什么情况下它能带来某些好处呢？条款 34 和 35 提供了这个问题的答案、以及其它更多讨论。

但首先请看这样一个问题。有一种“和机器类似”的编程结构，它在很多领域非常有用，而且在现实世界系统中频繁出现。从语言的角度来看，这种结构常常适于编写这样的函数：函数的返回值是一个指针，指向——它自身！这种编程结构是什么？如何写出那样一个看似古怪的递归函数声明？要知道答案，请看条款 32。

条款 32：递归声明

难度：6

你能写出一个函数，使其返回指向函数自身的指针吗？如果可以，解释你为什么想那样做。

(1) 什么是函数指针？如何使用它？

(2) 假设可以写出这么一个函数，它能返回一个指向自身的指针。这个函数同样可以返回一个指针，指向任何一个“和它具有相同原型”的函数。这种功能什么时候会有用？请解释。

(3) 有可能写出一个函数 `f()`，使其返回指向自身的指针吗？在下面这种很自然的使用方式下，它应该很有用：

```
// 例 32-3
//
// FuncPtr 是某种函数指针的 typedef,
// 这种函数具有和 f() 相同的原型
FuncPtr p = f();           // 执行 f()
(*p)();                  // 执行 f()
```

如果可能, 请演示如何实现。如果不可能, 说明为什么。

思考

函数指针：回顾

1. 什么是函数指针? 如何使用它?

一个对象指针可以让你动态地指向某种类型的对象, 同样, 一个函数指针可以让你动态地指向具有某种原型的函数。例如:

```
// 例 32-1
//

// 为一个输入参数为 double、返回值为 int 的函数
// 创建一个名为 FPDoubleInt 的 typedef
//
typedef int (*FPDoubleInt)( double );

// 使用它
//
int f( double ) { /* ... */ }
int g( double ) { /* ... */ }
int h( double ) { /* ... */ }

FPDoubleInt fp;
fp = f;
fp( 1.1 );      // 调用 f()
fp = g;
fp( 2.2 );      // 调用 g()
fp = h;
fp( 3.14 );     // 调用 h()
```

运用得好的话, 函数指针会给运行时期带来显著的灵活性。例如, 有这样一个标准 C 函数 `qsort()`, 它有一个输入参数是函数指针, 指向具有某种原型的比较函数 (comparison function); 这样, 调用者代码就可以提供一个定制的比较函数, 用以扩充 `qsort()` 的行为。

状态机简介

2. 假设可以写出这么一个函数，它能返回一个指向自身的指针。这个函数同样可以返回一个指针，指向任何一个“和它具有相同原型”的函数。这种功能什么时候会有用？请解释。

很多情形映入脑海，但最常见的例子是在实现一部状态机的时候。

简而言之，状态机（state machine）由一组“可能的状态”和那些状态之间的一组“合法转换”组成。例如，一部简单的状态机看起来可能像图 6 那样。

当我们处于状态 Start 时，如果接收到输入 a，我们将转换到状态 S2；或者，如果接收到输入 be，我们将转换到状态 Stop。除此之外的任何输入在状态 Start 都是不合法的。在状态 S2，如果接收到输入 see，我们将转换到状态 Stop；除此之外的任何输入在状态 S2 都是不合法的。对于这部状态机来说，仅有两个合法的输入流：be 和 asee。

为了实现一部状态机，有时候，将每一个状态写成一个函数就够了。所有状态函数都具有相同的原型，它们都返回一个指针，指向下一个将要调用的函数（状态）。例如，下面是一个极度简化的代码片段，演示了上面的设计思想。

```
// 例 32-2
//
StatePtr Start( const string& input );
StatePtr S2 ( const string& input );
StatePtr Stop ( const string& input );
StatePtr Error( const string& input ); // 错误状态
```

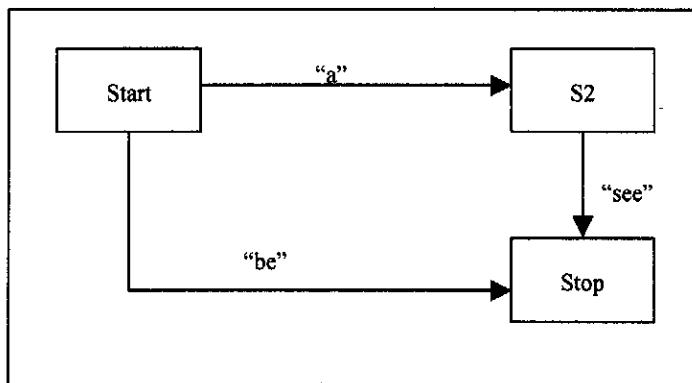


图 6：状态机示例

```

StatePtr Start( const string& input )
{
    if( input == "a" )
    {
        return S2;
    }
    else if( input == "be" )
    {
        return Stop;
    }
    else
    {
        return Error;
    }
}

```

关于状态机及其应用的更多介绍，请参阅你最喜爱的计算机科学教材。

当然，上面的代码没有说出 StatePtr 是什么；而且，想浅显地说出 StatePtr 是什么不一定那么容易。这恰好将我们引入到最后一个谜题中。

一个函数如何返回指向自身的指针？

3. 有可能写出一个函数 f()，使其返回指向自身的指针吗？在下面这种很自然的使用方式下，它应该很有用：

```

// 例 32-3
//
// FuncPtr 是某种函数指针的 typedef,
// 这种函数具有和 f() 相同的原型
FuncPtr p = f();           // 执行 f()
(*p)();                  // 执行 f()

```

如果可能，请演示如何实现。如果不可能，说明为什么。

是的，可能。但实现方法不是那样显而易见。

例如，有人会首先想到直接使用一个 typedef，写出下面这样的代码：

```

// 例 32-3(a): 天真的尝试（不正确）
//
typedef FuncPtr (*FuncPtr)(); // 错误

```

可惜，这种递归形式的 typedef 是不合法的。既然类型系统（type system）成为了障碍，一些人会采用迂回战术，通过在 void* 之间来回转型（casting）来绕过类型系统这个“障碍”：

```

// 例 32-3(b): 一个非标准且不具可移植性的有害手法
//
typedef void* (*FuncPtr)();
void* f() { return (void*)f; } // 转换为 void*

```

```
FuncPtr p = (FuncPtr)(f()); // 从 void*转换回来
p();
```

这不是一个解决方案，因为它不满足提问中的要求。更严重的是，这样做很危险，因为它是在有意攻击类型系统；这样做还会带来麻烦，因为它强迫 f() 的所有用户去进行类型转换。最糟糕的是，例 32-3(b) 的做法不符合标准。尽管一个 void* 的大小足以保存任何对象指针的值，但它不一定适合保存一个函数指针。在某些平台上，一个函数指针比一个对象指针要大。即使例 32-3(b) 那样的代码刚好在你目前使用的编译器上可行，但它不具可移植性；在另一个编译器上，甚至在你目前使用的编译器的下一个版本上，它可能行不通。

当然，有一个办法可以避免这个问题，我们可以借助另一种“函数指针类型”来回转型，而不是借助简单的 void*：

```
// 例 32-3(c)：符合标准且具可移植性，
//           但还是一个有害手法
//
typedef void (*VoidFuncPtr)();
typedef VoidFuncPtr (*FuncPtr)();

VoidFuncPtr f() { return (VoidFuncPtr)f; } // 转换为 VoidFuncPtr
FuncPtr p = (FuncPtr)f(); // 从 VoidFuncPtr 转换回来
p();
```

这段代码在技术上是合法的，但几乎和例 32-3(b)一样，它还是存在一个大问题：它是一个危险的有害手法，因为它在有意破坏类型系统。它将难以接受的负担强加给 f() 的所有用户，因为它要求用户必须进行类型转换。同样，它当然也不是一个真正的解决方案，因为它不满足提问中的要求。

真的不能做得更好吗？

一个正确且具可移植性的方案

幸运的是，要想完全达到问题 3 要求的效果，的确有一个类型安全且具可移植性的方案，它无须依靠非标准的代码，也无须依靠非类型安全的转换。其做法是：增加一层间接性——具体形式是一个代理类 (proxy class)，这个类不但接受它想要的指针类型，同时还有一个隐式转换，用以转换到它想要的指针类型：

```
// 例 32-3(d)：正确的方案
//
class FuncPtr_;
typedef FuncPtr_ (*FuncPtr)();
```

```

class FuncPtr_
{
public:
    FuncPtr_( FuncPtr p ) : p_( p ) { }
    operator FuncPtr() { return p_; }
private:
    FuncPtr p_;
};

```

现在，我们可以很自然地声明、定义和使用 f():

```

FuncPtr_ f() { return f; } // 自然的 return 语法

int main()
{
    FuncPtr p = f(); // 自然的使用语法
    p();
}

```

这个方案具有三大优点：

- (1) 它解决了提出的问题。更好的是，它不失类型安全，且具有可移植性。
- (2) 它的实现机制是透明的。对调用者或用户来说，使用它时采用的语法很自然；对函数自身的“return myname;”语句来说，语法也很自然。
- (3) 它的开销可能为零。在现代编译器上，借助内联和优化，代理类连同它的存储空间和成员函数会被消除得无影无踪。

结尾

通常情况下，你很可能想将这种专用的 FuncPtr_ 代理类（包含某个旧对象，而且不关心它的类型）模板化，使之成为一个通用的 Holder 代理类。可惜，对于上面这个 FuncPtr_ 类，你却不能直接将它模板化；如果那样做的话，typedef 看起来必然会像下面这样：

```
typedef Holder<FuncPtr> (*FuncPtr)();
```

它引用了自身。

条款 33：模拟嵌套函数

难度：5

C++ 有嵌套类，但没有嵌套函数。你必须何时有用的时候，在 C++ 中实现嵌套函数呢？

(1) 什么是嵌套类？它有什么用处？

(2) 什么是局部类？它有什么用处？

(3) C++不支持嵌套函数。也就是说，我们不能写出下面这样的代码：

```
// 例 33-3
//
int f( int i )
{
    int j = i*2;

    int g( int k ) // 不合法的 C++代码
    {
        return j+k;
    }

    j += 4;

    return g( 3 );
}
```

请示范：怎样做才有可能在标准 C++中得到相同的效果，并演示如何将解决方案扩充至一般化。

回顾

回顾：嵌套类和局部类

在信息隐藏和从属关系管理方面，C++提供了很多有用的工具。在回顾嵌套类和局部类这两个概念时，不要过多地纠缠于语法（syntax）和语义（semantics）。相反，要将注意力集中在：如何应用这些特性写出健壮、易于维护的代码，并表达出优良的对象设计——具有弱耦合性、强内聚性的设计。

1. 什么是嵌套类？它有什么用处？

嵌套类（nested class）是一个“包含在另一个类（范围）中”的类。例如：

```
// 例 33-1: 嵌套类
//
class OuterClass
{
/*...public, protected, 或 private...*/:
    class NestedClass
    {
        // ...
    };

    // ...
};
```

在组织代码、控制访问权限和从属关系方面，嵌套类很有用处。和类中的其它部分一样，嵌套类也遵守访问规则。所以，在例 33-1 中，如果 NestedClass 被声明为 public，那么，任何外部代码都可以称之为 OuterClass::NestedClass。通常，嵌套类包含的是私有实现细节，因而会被声明为 private。在例 33-1 中，如果 NestedClass 为 private，那么，只有 OuterClass 的成员（member）和友元（friend）可以使用 NestedClass。

注意，仅使用名字空间不可能达到与此相同的效果，因为名字空间只是将名称组织为不同的区域。名字空间本身不提供访问控制，但类会提供。所以，如果想对一个类的访问权限进行控制，一个可以使用的方法是：将它嵌套在另一个类中。

2. 什么是局部类？它有什么用处？

局部类（local class）是一个“定义在一个函数范围内”的类——任何函数，无论是成员函数还是自由函数。例如：

```
// 例 33-2: 局部类
//
int f()
{
    class LocalClass
    {
        // ...
    };

    // ...
}
```

像嵌套类一样，在管理代码的从属关系方面，局部类是一个有用的工具。在例 33-2 中，只有 f() 的内部代码知道 LocalClass 并能够使用它。当 LocalClass 是 f() 的内部实现细节、因而永远不应该公开给其它代码的时候，它就显示出了价值。

在可以使用非局部类的大多数地方，你也可以使用局部类；但有一个很重要的限制一定要牢记在心：局部类或未命名的类不能作为模板参数使用。以下摘自 C++ 标准 [C++98] 第 14.3.1/2 节：

局部类型、没有链接（译注：linkage，精确定义详见 C++ 标准 3.5 节）的类型、未命名的类型以及从这些类型中的任何一个复合而成（compounded）的类型都不能作为模板类型参数（template type-parameter）的实参（template-argument）使用。[示例：

```
template <class T>
class X { /* ... */ };
void f()
{
    struct S { /* ... */ };
    X<S> x3;      // 错误: 局部类型
                    // 用作模板实参
    X<S*> x4;      // 错误: 局部类型的指针
```

```
// 用作模板实参
}

--示例结束]
```

在 C++ 用于信息隐藏和从属关系管理的众多有用工具中，嵌套类和局部类是其中的两个。

嵌套函数：概述

一些语言（不包括 C++）支持嵌套函数（nested function），它和嵌套类有相似之处。嵌套函数定义在另一个函数——即外围函数（enclosing function）——之内，使得：

- 嵌套函数可以访问外围函数的变量；
- 嵌套函数局部存在于外围函数之内——即，除非外围函数提供一个指向嵌套函数的指针，否则我们无法在其它地方调用这个嵌套函数。

嵌套类之所以有用，在于它有助于控制一个类的可见性（visibility）；同样，嵌套函数之所以有用，在于它有助于控制一个函数的可见性。

C++ 中没有嵌套函数。那么，我们能够得到相同的效果吗？这为我们带来了下面这个重要问题。

3. C++ 不支持嵌套函数。也就是说，我们不能写出下面这样的代码：

```
// 例 33-3
//
int f( int i )
{
    int j = i*2;

    int g( int k ) // 不合法的 C++ 代码
    {
        return j+k;
    }

    j += 4;

    return g( 3 );
}
```

请示范：怎样做才有可能在标准 C++ 中得到相同的效果，并演示如何将解决方案扩充至一般化。

是的，有可能。只需要重新组织一下代码，只有一点小小的局限性。基本思想是将函数转换为函数对象。这个讨论也恰好显示了函数对象的某些威力，这一点并非巧合。

尝试在 C++ 中模拟嵌套函数

要解决问题 3 那样的难题，大多数人会从下面这样的尝试开始：

```
// 例 33-3(a): 幼稚的“局部函数对象”方案
//           (不正确)
//
int f( int i )
{
    int j = i*2;

    class g_
    {
public:
    int operator()( int k )
    {
        return j+k; // 错误: j 不能被访问
    }
} g;

j += 4;

return g( 3 );
}
```

例 33-3(a)的设计思想是：将函数包装在一个局部类中，并通过一个函数对象调用这个函数。

主意不错，但却不可行，原因很简单：这个局部类对象不能访问外围函数的变量。

“那么”，有人会说，“为什么不将函数中所有变量的指针或引用提供给这个局部类呢？”确实，通常这是下一步尝试。

```
// 例 33-3(b): 幼稚的“局部函数对象”
//           加变量的引用”方案,
//           (复杂、脆弱)
//
int f( int i )
{
    int j = i*2;

    class g_
    {
public:
    g_( int& j ) : j_( j ) {}

    int operator()( int k )
    {
        return j_+k; // 通过引用访问 j
    }
}
```

```

private:
    int& j_;
} g( j );
j += 4;
return g( 3 );
}

```

是的，我得承认这个方案“可行”——但只是勉强可行。这个方案脆弱、难以扩充；准确地说，我们只能将它看作一种非常手段。例如，要增加一个新变量，我们将要做四个改动：

- (a) 增加这个变量；
- (b) 为 `g_` 增加一个相应的私有引用成员；
- (c) 为 `g_` 增加一个相应的构造函数参数；
- (d) 为 `g_::g_()` 增加一个相应的初始化操作。

这将难以维护，而且也不容易扩充为多个局部函数。不能做得更好吗？

一个略有改善的方案

把变量本身转移到局部类中，我们就可以做得更好。

```

// 例 33-3(c): 一个更好的方案
//
int f( int i )
{
    class g_
    {
    public:
        int j;

        int operator()( int k )
        {
            return j+k;
        }
    } g;

    g.j = i*2;
    g.j += 4;

    return g( 3 );
}

```

这个方案向前迈出了坚实的一步。现在，在 `g_` 的数据成员中，指向外部数据的指针或引用不再需要；它也不需要构造函数；一切都更加自然。还请注意，这一技术现在可以扩充到任意数目的局部函数；所以，让我们稍微疯狂一点，多增加几个局部函数，譬

如 `x()`、`y()`、`z()` 等；与此同时，我们还趁机将 `g_` 换个名字，让它更能体现局部类的实际功能。

```
// 例 33-3(d): 快接近正确方案了!
//
int f( int i )
{
    // 定义一个局部类，包装
    // 所有的局部数据和函数
    //
    class Local_
    {
public:
    int j;

    // 所有的局部函数都放在这里:
    //
    int g( int k )
    {
        return j+k;
    }
    void x() { /* ... */ }
    void y() { /* ... */ }
    void z() { /* ... */ }
} local;

local.j = i*2;
local.j += 4;

local.x();
local.y();
local.z();

return local.g( 3 );
}
```

这还是有问题：当你需要用其它什么东西而不是缺省值来初始化 `j` 时，你必须为局部类增加一个蹩脚的构造函数，用以传递初始化值。在最初的提问中，我们是将 `j` 初始化为 `i*2` 的值。这里，我们不得不创建 `j` 然后对它赋值，这就和最初的要求不尽相同；而且，对于更复杂的类型，这会有困难。

一个完整的方案

如果你不需要让 `f` 本身成为一个“真正的”函数（例如，不需要取函数指针），你就可以将这一切转换为一个函数对象，并且可以非常漂亮地支持非缺省初始化。

```
// 例 33-3(e): 一个完整、极具
// 扩充性的方案
```

```

// 
class f
{
    int retval; // f 的“返回值”
    int j;
    int g( int k ) { return j + k; };
    void x() { /* ... */ }
    void y() { /* ... */ }
    void z() { /* ... */ }

public:
    f( int i ) // 最初的函数，现在是一个构造函数
        : j( i*2 )
    {
        j += 4;
        x();
        y();
        z();
        retval = g( 3 );
    }
    operator int() const // 返回结果
    {
        return retval;
    }
};

```

为了让代码更简短，我们以内联的形式书写代码，但所有私有成员也可以隐藏在一个 Pimpl 之后，从而像最初那个简单的函数一样，将接口同实现完全分离。

请注意，这个方案可以很容易地扩充为成员函数。例如，假设 f() 不是自由函数，而是个成员函数，我们想在 f() 中写一个嵌套函数 g()，如下所示：

```

// 例 33-4：这不是合法的 C++ 代码，但它演示
//           了我们的需要：一个局部函数存在
//           于一个成员函数之中
//
class C
{
    int data_;

public:
    int f( int i )
    {
        // 一个假想的嵌套函数
        int g( int i ) { return data_ + i; }

        return g( data_ + i*2 );
    }
};

```

要想表达这种关系，我们可以将 f() 变成一个类，就像例 33-3(e) 演示的那样；只不过，例 33-3(e) 中的那个类是在全局空间，而现在它是一个嵌套类，并需要通过辅助函数来访问。

```
// 例 33-4(a): 完整且极具可扩充性
//           的方案，现在应用于
//           成员函数
//
class C
{
    int data_;
    friend class C_f;
public:
    int f( int i );
};

class C_f
{
    C* self;
    int retval;
    int g( int i ) { return self->data_ + i; }

public:
    C_f( C* c, int i ) : self( c )
    {
        retval = g( self->data_ + i*2 );
    }

    operator int() const { return retval; }
};

int C::f( int i ) { return C_f( this, i ); }
```

总结

在上面例 33-3(e) 和 33-4(a) 中，我们所演示的方法模拟了局部函数的大多数特性，而且，我们还可以很容易地将其扩充到任意数目的局部函数。其主要不足在于，它要求所有变量都被定义在“函数”的开始处，不（轻易地）允许我们将变量声明放在离“第一次使用变量”更近的地方。和局部函数相比，它写起来更让人感到枯燥，它显然只是针对语言限制的权宜之计。但归根结底，它还不错；它演示了函数对象相对于普通函数所具有的与生俱来的威力。

本条款的目的并不在于让你去认识“C++ 中有了局部函数会有多好”，一如既往，本条款的目的在于：先提出一个特定的设计问题，然后探究解决问题的各种方案，最后权衡这些方案并作出最佳选择。沿着这条道路，我们也体验了各种不同的 C++ 特性，透彻理解了函数对象在应用中威风八面的原因。

在设计自己的程序时，无论如何都要力求简化和整洁。上面的一些过渡方案的确“可行”，但在你的产品代码中，你绝对不能让它们得见天日。它们复杂、难以理解，因而使得维护更加困难、更加昂贵。

设计准则

力求清晰。避免复杂的设计。避免招致困惑。

简单的设计更易于实现和测试。一定要避免复杂的方案；几乎可以肯定，复杂的方案更脆弱、更难以理解和维护。在小心谨慎地避免设计过度的陷阱时，请认识到：即使找出一个简单合理的设计会让你在前面多花一点时间，但它所带来的长期好处通常会让你觉得这是值得的。现在多花点时间往往意味着节省“以后更多的时间”。大多数人一定会赞成：“以后更多的时间”最好是和家人在乡间别墅度过，而不是趴在键盘上，在像老鼠窝一样凌乱的代码中去寻找那最后几只臭虫。☺

条款 34：预处理宏

难度：4

在 C++ 从 C 继承的遗产中，预处理器宏是其中的一部分。本条款讨论：在现代 C++ 的发展过程中，预处理器宏是否还有意义。

C++ 中有那么多灵活的特性，例如重载、类型安全的模板，等等；那么，为什么 C++ 程序员还要写“#define”这样的预处理指令？

解答

有了 C++ 特性，我们常常（但并非总是）会否决掉对#define 的需要。例如，“const int c = 42;”就比“#define c 42”更佳，因为它提供了类型安全、避免了预处理器的意外修改，此外还有其它很多好处。

然而，还是有一些很好的理由去使用#define。

1) 守护头文件

为了防止头文件被多次包含，这是一种常用技巧。

```
#ifndef MYPROG_X_H
#define MYPROG_X_H
```

```
// ... 头文件 x.h 的其余部分...
#endif
```

2) 使用预处理器特性

在诊断代码中，插入行号或编译时间这类信息通常很有用。要想做到这一点，一个简单的方法是使用预定义的标准宏，例如 `_FILE_`、`_LINE_`、`_DATE_` 和 `_TIME_`。基于相同原因，以及其它原因，使用 stringizing（字符串化）和 token-pasting（标记合并）预处理运算符（#和##）也很有用。

3) 在编译时期选择代码（或称：特定编译代码）

预处理使用多了，你会觉得这是它的一种很重要的使用方式。尽管我绝不是预处理技术的狂热分子，但我也知道，有些事情用别的方法很难做到，甚至根本做不到。

A. 调试代码

在编译你的系统时，有时你想使用某些“额外”代码（通常是一些调试信息），但有时你又不想这样做。

```
void f()
{
    #ifdef MY_DEBUG
        cerr << "some trace logging" << endl;
    #endif

    // ... f() 的其余部分...
}
```

这段代码实际上让我们有了两个不同的程序。如果在编译时定义了 `MY_DEBUG`，我们得到的是包含“输出到 `cerr`”的源代码，编译器也会去检查那行代码的语法和语义。如果在编译时没有定义 `MY_DEBUG`，我们得到的是不同的程序，它不包含那行“输出到 `cerr`”的代码；对于那行没有包含在内的代码，编译器也就无法检查其正确性。

通常，用条件表达式来代替这个`#define`会更好。

```
void f()
{
    if ( MY_DEBUG )
    {
        cerr << "Some trace logging" << endl;
    }
}
```

```
// ...f() 的其余部分...
}
```

采用这种方式，我们只用对一个程序进行编译和测试，编译器会检查所有的代码；如果 `MY_DEBUG` 不为 `true`，编译器会很容易地忽略掉无法到达的代码。

B. 特定平台代码

通常，在处理“针对特定平台”的代码时，最好运用 `factory` 模式；采用这种方法，代码的组织会更合理，运行时期会更具灵活性。但有时，由于存在的差异太少，你很难去构造一个合理的 `factory`，这时候，预处理就是一种切换可选代码的好办法。

C. 不同的数据表示方式

一个常见的例子是：对于一个模块所定义的一组错误代码，外部用户看到的应该是一个简单的 `enum`，并带有注释；但在模块内部，它们应该被存储在一个 `map` 中，以便于查找。即：

```
// 对于外部用户
//
enum Error
{
    ERR_OK = 0,           // No error
    ERR_INVALID_PARAM = 1 // <description>
    ...
}

// 对于模块的内部使用
//
map<Error, const char*> lookup;
lookup.insert( make_pair( ERR_OK,
                         (const char*)"No error" ) );
lookup.insert( make_pair( ERR_INVALID_PARAM,
                         (const char*)<description>" ) );
...
```

我们想同时拥有两种表示方式，但不希望将实际信息（成对的“代码/消息”）定义两次。有了宏这一魔法，我们就可以像下面这样，简单地写一个错误列表，在编译时期创建相应的数据结构：

```
ERR_ENTRY( ERR_OK, 0, "No error" ),
ERR_ENTRY( ERR_INVALID_PARAM, 1, "<description>" ),
...
```

`ERR_ENTRY` 以及相关的宏的实现留给读者去完成。

这里只是列出了三个常见的例子，除此之外还有很多应用。虽然在很多地方，我们应该避免使用 C 风格的预处理，但预处理还是有其自身的价值；明智地使用它，我们写出的 C++ 代码就会更简单、更安全。

设计准则

除了以下情况之外，避免使用预处理宏：

- 守护头文件；
- 条件编译，以获得可移植性，或在.cpp 文件（不是.h 文件！）中进行调试；
- 用#pragma 禁止掉无伤大雅的警告，但这种#pragma 总得包含在一个“为了获得可移植性而提供的条件编译”之中，以防编译器不认识它们而发出警告。

要想知道 C++ 的发明者如何看待预处理，请阅读[Stroustrup94]第 18 章。

条款 35：宏定义

难度：4

宏能够做什么？不能够做什么？不是所有的编译器都对你惟命是从。

- (1) 示范如何写一个简单的预处理宏 max()；这个宏取两个参数，通过普通的“<”运算，它比较出其中的较大值。在写这样一个宏时，一般会有哪些易犯错误？
- (2) 预处理宏不能创建哪些东西？为什么不能？

解答

宏的常见陷阱

1. 示范如何写一个简单的预处理宏 max()；这个宏取两个参数，通过普通的“<”运算，它比较出其中的较大值。在写这样一个宏时，一般会有哪些易犯错误？

有四大易犯错误；除此之外，宏还有几个缺陷。我们首先看看这些易犯错误；在写一个宏时，下面是经常出错的地方。

1) 不要忘记为参数加上括号

```
// 例 35-1(a): 括号陷阱之一: 参数
//
#define max(a,b) a < b ? b : a
```

这里的问题在于，使用参数 a 和 b 时没有完全加上括号。宏只作直接的文本替换，所以上面那种做法会造成不可预料的后果。例如：

```
max( i += 3, j )
```

展开后成为

```
i += 3 < j ? j : i += 3
```

考虑到运算符优先级和语言规则，它实际上是

```
i += ((3 < j) ? j : i += 3)
```

这种错误可能需要长时间的调试才能发现。另一个问题和顺序点（sequence points）有关，在不同的顺序点，它会对 i 修改两次；关于这个问题，一会儿再作详细说明。

2) 不要忘记为整个展开式加上括号

解决了第一个问题，我们又身陷另一个机关。

```
// 例 35-1(b): 括号陷阱之二: 展开式
//
#define max(a,b) (a) < (b) ? (b) : (a)
```

现在的问题是，整个展开式没有被正确地加上括号。例如：

```
k = max( i, j ) + 42;
```

展开后成为

```
k = (i) < (j) ? (j) : (i) + 42;
```

考虑到运算符优先级，它实际上是

```
k = (((i) < (j)) ? (j) : ((i) + 42));
```

如果 $i \geq j$ ，k 被赋以 $i+42$ 的值，这一点如你所愿；但是，如果 $i < j$ ，k 被赋以 j 的值。

我们可以为宏的整个展开式加上括号，从而解决第二个问题，但还有另外一个问题：

3) 当心多参数运算

如果其中某个表达式有副作用，或者两个都有副作用，想想会发生些什么：

```
// 例 35-1(c): 多参数运算
//
#define max(a,b) ((a) < (b) ? (b) : (a))
max( ++i, j )
```

如果`++i`的结果大于等于`j`，`i`会递增两次，这可能不是程序员想要的：

```
((++i) < (j) ? (j) : (++i))
```

类似地，请看

```
max( f(), pi)
```

它展开后成为：

```
((f()) < (pi) ? (pi) : (f()))
```

如果`f()`的结果大于等于`pi`，`f()`会执行两次；几乎可以肯定，这绝对缺乏效率，而且往往是错误的。

虽然头两个问题可以解决，但这是一块真正的硬石头。只要`max`是个宏，我们对此只能束手无策。

4) 名字冲突

还有一点：宏对“范围（scope）”漠不关心。（宏对大多数事都漠不关心；参见[GotW]#63。^④）它只是执行文本替换，而不管文本在哪儿。这意味着，只要使用宏，我们就得小心地对这些宏命名。具体来说，这个`max`宏最大的问题是，它极有可能会和标准的`max()`函数模板相冲突：

```
// 例 35-1(d): 名字冲突
//
#define max(a,b) ((a) < (b) ? (b) : (a))
#include <algorithm> // 噢！
```

问题在于，在头文件`<algorithm>`中，有像下面这样的东西：

```
template<typename T> const T&
max(const T& a, const T& b);
```

于是，宏“好心好意”地将它转换成一团糟，从而导致无法编译：

^④ 在线提供于 <http://www.gotw.ca/gotw/063.htm>。

```
template<typename T> const T&
((const T& a) < (const T& b) ? (const T& b) : (const T& a));
```

如果你认为，只需要将宏定义放在所有被包含的头文件之后（在任何情况下这都是一个好的做法），这种问题就很容易避免，那么请想象一下，如果在其它代码中，有些变量或什么东西刚好也叫 max，宏会对那些代码做些什么好事呢？

如果你非得写一个宏不可，那么，尽量为它想出一个不平常的、难以拼写的名字，这样才能最大可能地避免和其它名字相冲突。

宏的其它缺陷

还有一些重要的事情宏无法做到：

5) 宏不能递归

我们可以写出一个递归函数，但不可能写出一个递归的宏。正如 C++ 标准[C++98] 第 16.3.4/2 节所述：

在本次替换表扫描期间，如果发现了正在被替换的宏的名称（不包括源文件中其余的预处理标记），这个名称将不被替换。此外，在任何嵌套替换中，如果碰到了正在被替换的宏的名称，这个名称也不会被替换。这些未被替换的宏的名称（预处理标记）不再用于进一步替换，即使后来在某些环境下（又）检查到这些本该已经被替换掉的宏的名称（预处理标记）。（译注：C++ 标准第 16.3.4 节所描述的是再次扫描及进一步替换的规则。要准确理解这段文字，请参阅原文上下文。）

6) 宏没有地址

你有可能得到任何自由函数或成员函数的指针（例如，把它用作 predicate），但你不可能得到一个宏的指针，因为宏没有地址。宏之所以没有地址，原因很显然——宏不是代码。宏不会以自身的形式存在，因为它只是一种被美化了的（不是特别光荣的）文本替换规则。

7) 宏有碍调试

在编译器有机会看到代码之前，宏就会修改相应的代码——因而，它会严重破坏变量名称和其它名称；此外，在调试阶段，你无法跟踪到宏的内部。

你听说过这样一个故事吗？——科学家开始拿律师做试验，而不是实验室的宏。◎这是因为……

有些事甚至宏也不会去做

使用宏有许多正当的理由（详细介绍请参见条款 34），但使用宏也有限制。这给我们带来最后一个问题。

2. 预处理宏不能创建哪些东西？为什么不能？

在 C++ 标准中，条款 2.1 对编译阶段（phases of translation）作了严格规定。预处理指令和宏的展开发生在第 4 个阶段。这样一来，在一个符合标准的编译器上，宏不可能创建以下任何东西：

- 三图符（trigraph）（三图符在第 1 个阶段被替换）；
- 通用字符名（\uXXXX，在第 1 个阶段被替换）；
- 用于行连接的行尾反斜线符（在第 2 个阶段被替换）；
- 注释（在第 3 个阶段被替换）；
- 另一个宏或预处理指令（在第 4 个阶段展开和执行）；
- 通过字符串中宏的名称改变字符字面值（character literal，例如，'x'）或字符串字面值（string literal，例如，"hello, world"）。

关于最后一点，C++ 标准第 16.3/8 节的脚注 7 有如下说明：

在宏替换期间，所有字符字面值和字符串字面值都是预处理标记，而不是“可能包含标识符似的子序列”的序列（参见 2.1.1.2，编译阶段），所以，它们永远不会被作为“宏的名称”或“参数”来扫描。

一篇公开发表的文章^①声称，宏有可能创建注释，像下面这样：

```
#define COMMENT SLASH(/)
#define SLASH(s) /##s
```

这种做法不符合标准、不具有可移植性；但这是一个可以理解的错误，因为它的的确在某些很普及的编译器上可行。为什么可行？因为那些编译器没有正确地支持“编译阶段”。

^① M. Timperley, “A C/C++ Comment Macro” (C/C++ Users Journal, 19(1), 2001 年 1 月)。

杂项议题

一些议题很重要，但不便于分门别类。在本书的最后五个条款中，我们的讨论将集中在四个语言特性上：初始化、前置声明、`typedef`，最后是有关名字空间的两个讨论。特别是，条款 39 和 40 针对一个经常提出的问题作出了回答，然后结束全书；那是一个有关现代 C++ 的常见问题：到底该在何时使用名字空间？如何使用？让我们开始吧！

条款 36：初始化

难度：3

直接初始化和拷贝初始化之间有什么区别？分别在何时使用它们？

(1) 直接初始化和拷贝初始化有何不同？

(2) 在下面的例子中，哪些地方使用了直接初始化，哪些地方使用了拷贝初始化？

```
class T : public S
{
public:
    T() : S(1),           // 基类初始化
          x(2) {}         // 成员初始化
    X x;
};

T f( T t )           // 传递函数参数
{
    return t;           // 返回值
}

S s;
T t;
```

```

S& r = t;

reinterpret_cast<S&>(t); // 执行 reinterpret_cast
static_cast<S>(t);      // 执行 static_cast
dynamic_cast<T&>(r);   // 执行 dynamic_cast
const_cast<const T&>(t); // 执行 const_cast

try
{
    throw T();           // 抛出异常
}
catch( T t )           // 处理异常
{
}

f( T(s) );             // 函数形式的类型转换
S a[3] = { 1, 2, 3 };  // 大括号初始化语句
S* p = new S(4);       // new 表达式

```

解答**1. 直接初始化和拷贝初始化有何不同？**

直接初始化 (direct initialization) 指的是：使用单个构造函数（可能是转换构造函数）来初始化对象；它相当于 “`T t(u);`” 的形式：

```

U u;
T t1(u); // 调用 T::T( U& ), 或类似

```

拷贝初始化 (copy initialization) 指的是：使用拷贝构造函数来初始化对象，如果必要，在此之前会先调用一个用户自定义的转换；它相当于 “`T t=u;`” 的形式：

```

T t2 = t1; // 相同类型：调用 T::T( T& ), 或类似
T t3 = u;  // 不同类型：调用 T::T( T(u) )
            // 或 T::T( u.operator T() ), 或类似

```

旁白：之所以加上“或类似”这样含糊的用词，原因是：拷贝和转换构造函数可以接受的参数可能会和普通引用稍有不同（即，这个引用可能为 `const`，或 `volatile`，或二者兼备）；此外，用户自定义的转换构造函数或运算符还可以接受和返回一个对象，而不是引用；而且，拷贝和转换构造函数还可以有缺省参数。

注意，在最后一个例子 “`T t3 = u;`” 中，编译器可以调用用户自定义的转换（以构造一个临时对象）和 `T` 的拷贝构造函数（以从临时对象构造 `t3`）；或者，也可以省略临时对象，直接从 `u` 构造 `t3`（这就最终等价于 “`T t3(u);`”）。无论哪种方法，未被优

化的代码还是得合法。具体来说，拷贝构造函数还是得可以访问——即使对它的调用会被优化掉。

和 C++ 标准出台之前相比，如今，编译器省略临时对象的自由度有了限制。但对于上面这种优化、乃至返回值优化（return value optimization），省略还是允许的。要想了解这方面的更多信息，请阅读 *Exceptional C++* [Sutter00]；在那本书中，条款 42 是这方面的基本介绍，条款 46 则对 C++ 标准最终所作的修改进行了说明。

设计准则

变量初始化时，尽量采用“`T t(u)`”形式，不要采用“`T t = u`”形式。

2. 在下面的例子中，哪些地方使用了直接初始化，哪些地方使用了拷贝初始化？

在 C++ 标准第 8.5 节，你可以找到下面大多数示例的答案。此外，这里还安排了三个圈套让你去钻，因为它们完全没有涉及到初始化。你注意到了吗？

```
class T : public S
{
public:
    T() : s(1),           // 基类初始化
          x(2) {}         // 成员初始化
    x x;
};
```

基类和成员的初始化都是采用直接初始化。

```
T f( T t )           // 传递函数参数
{
    return t;          // 返回值
}
```

值的传递和返回都是采用拷贝初始化。

```
S s;
T t;
S& r = t;

reinterpret_cast<S&>(t);      // 执行 reinterpret_cast
dynamic_cast<T&>(r);        // 执行 dynamic_cast
const_cast<const T&>(t);     // 执行 const_cast
```

圈套：这些地方完全没有涉及到新对象的初始化，只是创建了引用。

```
static_cast<S>(t);           // 执行 static_cast
static_cast 使用直接初始化。
```

```

try
{
    throw T();           // 抛出异常
}
catch( T t )          // 处理异常
{
}

```

异常对象的抛出和捕获都是使用拷贝初始化。

注意，上面这段代码存在两个 T 对象的拷贝，因而一共有三个 T 对象。在异常抛出现场，被抛出对象的拷贝会被创建。另外，在代码的异常处理程序中，被抛出对象是通过传值来捕获的，所以还会创建第二个拷贝。

但在一般情况下，我们应该尽量通过“引用”来捕获异常，而不是通过“值”；这样，我们就可以避免创建额外的拷贝，并消除可能存在的对象切割（object slicing）。^④

```
f( T(s) );           // 函数形式的类型转换
```

这种“函数风格的类型转换”采用的是直接初始化。

```
s a[3] = { 1, 2, 3 };      // 大括号初始化语句
```

大括号初始化语句采用的是拷贝初始化。

```
s* p = new S(4);        // new 表达式
```

最后，new 表达式采用的是直接初始化。

条款 37：前置声明

难度：3

前置声明是“消除不必要的编译时期依赖性”的一种重要方法。但在这个例子中，我们演示的却是一个有关前置声明的陷阱。你将如何避免它？

(1) 前置声明（forward declaration）是非常有用的工具。但在这个例子中，它没有像程序员所预计的那样工作。做了标记的那两行代码为什么是错误的？

```

// 文件 f.h
//
class ostream; // 错误
class string; // 错误
string f( const ostream& );

```

^④ 参见[Meyers96]条款 13.

(2) 不包含任何其它文件，为上面的 `ostream` 和 `string` 写出正确的前置声明。

解答

1. 前置声明（forward declaration）是非常有用的工具。但在这个例子中，它没有像程序员所预计的那样工作。做了标记的那两行代码为什么是错误的？

```
// 文件 f.h
//
class ostream; // 错误
class string; // 错误
string f( const ostream& );
```

错误出在：你不能以这种方式前置声明 `ostream` 和 `string`，因为它们不是类。它们都是模板的 `typedef`。

（是的，在 C++ 标准时出台之前，你确实可以用这种方式前置声明 `ostream` 和 `string`，但那已经是多年以前的事了。在标准 C++ 语言中，那已成过眼烟云。）

2. 不包含任何其它文件，为上面的 `ostream` 和 `string` 写出正确的前置声明。

简短的答案是：这不可能。实际情况是：不存在某种标准且具有可移植性的方法，可以做到不包含另一个文件却能前置声明 `ostream`；根本没有一种标准且具有可移植性的方法可以前置声明 `string`。

我们之所以不能对二者进行前置声明，原因在于，C++ 标准已经有了明确的规定，我们不能对 `namespace std` 写出我们自己的声明，而 `ostream` 和 `string` 就在 `namespace std` 之中：

除非另有说明，如果一个 C++ 程序向“`namespace std`”或“`namespace std` 内的某个名字空间”增加声明或定义，其后果不可预测。

除此之外，这条规定还允许：在供应商提供的标准库实现中，标准库模板具有的模板参数可以比标准所要求的还要多（当然，必须提供合适的缺省值，以保持兼容性）。但即使我们可以前置声明标准库模板和类，这种做法也不具有可移植性，因为供货商可以扩充那些声明，使得不同的实现都各不相同。

你能做得最好的（但它不是“不包含任何其它文件”这一问题的解决方案）只能是像下面这样：

```
#include <iostream>
#include <string>
```

`iosfwd` 头文件的确包含的是真正的前置声明。`string` 头文件则不是。这是你能做到的最好结果，并且不失可移植性。幸运的是，在实际工作中，前置声明 `string` 和 `ostream` 算不上什么大不了的事，因为这些头文件通常很小而且使用广泛。大多数标准头文件也都是这样。但请当心陷阱：不要经不起诱惑，试图前置声明那些属于 namespace `std` 中的模板或其它任何东西。这些至高无上的特权是保留给编译器和程序库的设计者——而且，仅仅是他们。

设计准则

当前置声明可以满足需要时，绝对不要包含 (`#include`) 头文件。在不需要流 (stream) 的完整定义时，尽量只包含 (`#include`) `<iosfwd>`

此外，请阅读 *Exceptional C++* [Sutter00] 条款 26。

条款 38: `typedef`

难度：3

为什么要使用 `typedef`？除了很多常规理由之外，我们还将看到另外一些 `typedef` 技术。它能让我们更安全更容易地使用 C++ 标准库。

(1) 为什么要使用 `typedef`？尽可能多地列举出使用它的场合和理由。

(2) 在用到了标准 (STL) 容器的代码中，使用 `typedef` 为什么是个好主意？

解答

Typedef：控制名字之中有什么

简单回顾：写下 `typedef`，你就可以给某个类型赋上另一个具有相同意义的名称。例如：

```
typedef vector< vector<int> > IntMatrix;
```

有了它，你就可以在代码中用更简单的 `IntMatrix` 取代冗长的 `vector< vector<int> >`。

1. 为什么要使用 `typedef`？尽可能多地列举出使用它的场合和理由。

这是 `typedef` 的几项“本领”：

便于打字

名字越简短，打字越容易。

可读性

`typedef` 能使代码更易于阅读，对长的模板类型名称来说，尤其如此。举一个简单的例子，在一个公共新闻组，曾经有一张贴子询问下面代码的含义：

```
int (*t(int))(int*);
```

如果你习惯于阅读 C 风格的声明，就像习惯于阅读一部维多利亚时代的英语小说（也就是那些密密麻麻冗长乏味的散文，读起来有时候觉得像是懒婆娘的裹脚布）一样，那么，你会知道答案、你会觉得没问题。如果不是这样，`typedef` 的确可以帮助你，即使是使用一个像 `Func` 这样没什么含义的 `typedef` 名称：

```
typedef int (*Func)(int*);  
Func t(int);
```

现在很清楚了，这是一个函数声明：此函数的名称为 `t`，输入参数为 `int`，返回值为一个函数指针，指向一个输入参数为 `int*`、返回值为 `int` 的函数。（请快速把这句话念三遍。②）在这种情况下，`typedef` 比语言文字更具可读性。

使用 `typedef` 还可以增加语义。例如，`PhoneBook` 比 `map<string, string>` 更容易理解；后者几乎没有语义，因而几乎可以表示任何东西。

便于交流

`typedef` 有助于表达程序的意图。例如，如果不在程序中四处乱扔不加修饰的 `int`，而是给这个类型一个名称——即使它本质上还是一个 `int`，那么，代码就会更清晰。请看下面这段代码：

```
int x;  
int y;  
y = x * 3; // 可能是正确的——谁知道？
```

再比较一下这段代码：

```
typedef int Inches;  
typedef int Dollars;  
  
Inches x;  
Dollars y;  
y = x * 3; // 嗯……？
```

如过你是程序的维护者，你喜欢读哪一段代码？

可移植性

对那些“和平台相关的名称”或其它“不具可移植性的名称”，如果我们使用 `typedef`，将它们移植到新的平台就会更容易。毕竟，如果这么写：

```
#if defined USING_COMPILER_A
    typedef __int32 Int32;
    typedef __int64 Int64;
#elif defined USING_COMPILER_B
    typedef int     Int32;
    typedef long long Int64;
#endif
```

你就不会为了某个“和平台相关的名称”而在整个代码中忙于查找和替换。有了 `typedef` 的帮助，你的代码就可以避免一些简单的平台依赖性问题。（但是，为达到这一目的，你一般不会使用 `#define`，原因参见条款 35.）

2. 在用到了标准（STL）容器的代码中，使用 `typedef` 为什么是个好主意？

灵活性

较之在整个代码中修改所有的名称，仅在一处修改 `typedef` 名称会更容易。例如，请看下面的代码：

```
void f( vector<Customer>& custs )
{
    vector<Customer>::iterator i = custs.begin();
    ...
}
```

假如几个月之后，我们最终发现使用 `vector` 容器不合适，那该怎么办？如果我们要存储巨量的 `Customer` 对象，`vector` 连续存储的特点就成为了一种缺点，我们就会转而使用 `deque`^④。或者，如果我们发现，我们需要在序列的中部频繁地插入和删除元素，我们就会转而使用 `list`。

那么，针对上面的代码，在出现类型名称 `vector<Customer>` 的所有地方，我们就都得作修改。但如果我们的代码是这样写的，一切该会多么容易：

```
typedef vector<Customer> Customers;
typedef Customers::iterator CustIter;

...
void f( Customers& custs )
{
```

^④ 参见条款 7.

```
CustIter i = custs.begin();
...
}
```

要作的改动仅仅只是将那个 `typedef` 变成 `list<Customer>` 或 `deque<Customer>`！当然，事情不会总是那么容易——例如，我们的代码可能需要 `Customers::iterator` 是个随机访问迭代器，而 `list<Customer>::iterator` 不是。尽管如此，`typedef` 还是可以让我们从一大堆枯燥的修改工作中解脱出来，但如果不用 `typedef`，这些工作就无法避免。

便于使用 traits

`traits` 技术具有强大的功能，它可以将信息和类型关联起来；如果你想定制标准容器或算法，你会经常需要提供 `traits`。有关 `traits` 的论述，请阅读本书“泛型程序设计”章节；此外，你还可以参阅 *Exceptional C++* [Sutter00] 条款 2 和 3，那里有一个不分大小写的 `string` 示例，我们定义了自己的 `char_traits`，用以替换其缺省版本。

总结

`typedef` 的应用大多逃脱不了以上几大类。一般来说，`typedef` 可以让代码更易于编写、更易于阅读、更易于通过众所周知的“额外一层间接”而被修改——在这里，这仅仅只是一种名称上的间接性，但它还是为你带来了好处。

条款 39：名字空间，之一：using 声明和 using 指令

难度：2

标准 C++ 支持名字空间，并通过 `using` 声明和 `using` 指令控制名称的可见性。本条款讨论这一主题，并为了一条新的实现应用做准备。

什么是 `using` 声明和 `using` 指令？如何使用它们？给出例子。另外，它们中的哪一个和顺序相关？

解答

在标准化的进程中，虽然 C++ 的变化很大，但标准委员会所做的修改极少会影响到现有的 C++ 代码。然而，有这样一个变化，它会使几乎所有以前的 C++ 代码不能通过编

译——这就是：名字空间这一特性的增加；特别是，如今，整个 C++ 标准库都存在于标准名字空间 std 之中。

如果你是一名 C++ 程序员，即使至今你还没碰到这方面的麻烦，那么，麻烦也会很快找上门；因为，大多数编译器的最新版本已经顺应了这一变化。在你开始将自己的编译器升级到最新版本时，或者，当你想停止使用不支持名字空间的旧式（如今不再符合标准的）头文件时，你的代码也不得不顺应这一变化。

什么是 using 声明和 using 指令？如何使用它们？给出例子。另外，它们中的哪一个和顺序相关？

Using 声明

using 声明（using-declaration）为实际声明在另一个名字空间中的名称创建一个本地同义词（local synonym）。为了重载和名称解析的需要，using 声明的运作方式和其它任何声明一样。

```
// 例 39-1(a)
//
namespace A
{
    int f( int );
    int i;
}

using A::f;

int f( int );

int main()
{
    f( 1 );           // 歧义：A::f() 还是 ::f()？

    int i;
    using A::i;      // 错误，重复声明（就像写
                      // 了两次 “int i;” 一样）
}
```

只有已经出现过声明的名称，using 声明才能将其引入。例如：

```
// 例 39-1(b)
//
namespace A
{
    class X {};
    int Y();
    int f( double );
}
```

```

using A::X;
using A::Y;           // 函数 A::Y(), 而非类 A::Y
using A::f;           // A::f(double), 而非 A::f(int)

namespace A
{
    class Y {};
    int f( int );
}

int main()
{
    X x;             // 正确，x 是 A::x 的同义词

    Y y;             // 错误，A::Y 不可见，因为
                     // using 声明出现在它想要
                     // 的实际声明之前。

    f( 1 );          // 哟：这里偷偷地使用了一
                     // 个隐式转换，它调用的是
                     // A::f(double)，而不是
                     // A::f(int)。因为在 using
                     // 声明之前，只有头一个
                     // A::f() 声明出现过。
}

```

这一特点使得 using 声明具有顺序相关性——特别是，如果一个名字空间分布在一组头文件中，在使用这个名字空间中的名称时，情况更是如此。头文件包含的顺序，特别是，哪个头文件包含在 using 声明之前或之后，都会影响到 using 声明会将哪些名称引入到所在空间。当然，名字空间 std 就恰好属于“一个名字空间分布在一组头文件中”的情况。关于这一点的更详细的讨论，请阅读下一条款。

using 指令

有了 using 指令（using-directive），另一个名字空间中的所有名称就都可以使用于 using 指令所在的空间。和 using 声明不同，using 指令会将声明在 using 指令之前、之后的所有名称都引入进来。当然，在可以使用某个名称之前，那个名称的声明还是得先出现过才行。例如：

```

// 例 39-1(c)
//
namespace A
{
    class X {};
    int f( double );
}

```

```

void f()
{
    X x;           // 正确, X 是 A::X 的同义词
    Y y;           // 错误, Y 没有出现过
    f( 1 );        // 正确, 调用 A::f(double),
}                  // 有参数类型提升 (parameter promotion)
                   // (译注: 关于 promotion, 请参考
                   // [Stroustrup00]附录 C.6.1)

using namespace A;

namespace A
{
    class Y {};
    int f( int );
}

int main()
{
    X x;           // 正确, X 是 A::X 的同义词
    Y y;           // 正确, Y 是 A::Y 的同义词
    f( 1 );        // 正确, 调用 A::f(int)
}

```

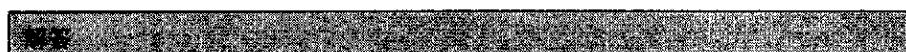
条款 40：名字空间，之二：迁徙到名字空间

难度：4

既想使用 C++ 威力强大的名字空间管理功能，又要避免其中的陷阱和易犯错误，如何做到最好？此外，要想将现有 C++ 代码移植到支持名字空间的编译器和程序库上，最有效的途径是什么？

假设你正在开发一个有数百万行代码的项目，其中的.h 和.cpp 文件多达上千。这时候，项目小组需要将编译器升级到最新版本；这个新版本的编译器支持名字空间，标准库的所有构件也都放在名字空间 std 中（或者，你的编译器早就支持名字空间，但直至现在，你的小组还在使用像<vector.h>这种不支持名字空间的旧式头文件）。不幸的是，这种顺应标准的做法（或头文件的迁移）有副作用，它会导致现有代码不能通过编译。和往常一样，你绝没有足够的时间去精耕细作。你当然想对每一个源文件仔细分析，然后为它们一一添上所需的 using 声明；但目前，时间不允许你这样做。

如何处理这一问题并将你的代码库安全地移植到新的（更符合标准的）环境？最有效的方式是什么？请讨论各种可能，并选择一个可以尽快完成工作的方案，同时不影响将来安全性和使用性。怎样才能将（对现在来说）不必要的移植工作最大限度地推迟到将来、同时不会增加以后移植工作的任务量？



安全高效地迁徙到名字空间

提问中描述的情况很典型，譬如：你想将项目转移到某个编译器，而这个编译器刚刚增加了对名字空间的支持；或者，你想在项目中抛弃不支持名字空间的旧式头文件，转而使用支持名字空间的头文件。这里之所以会有问题，并且需要一个解决方案，原因在于：如今，标准库存在于名字空间 `std` 之中；所以，在这个项目中，不带修饰地使用 `std::` 名称，代码将无法通过编译。

举例来说，下面的代码过去可以通过编译：

```
// 例 40-1：这段代码过去可以通过编译
//
#include <iostream.h> // “遗留下来的”标准出台之前的头文件

int main()
{
    cout << "hello, world" << endl;
}
```

现在，要么，你必须明确地写出哪些名称存在于 `std` 之中：

```
// 例 40-2(a)：选择 A，明确指明一切
//
#include <iostream>

int main()
{
    std::cout << "hello, world" << std::endl;
}
```

要么，使用 `using` 声明，将需要的 `std` 名称引入到当前空间：

```
// 例 40-2(b)：选择 B，使用 using 声明
//
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    cout << "hello, world" << endl;
}
```

要么，简单地使用一个 `using` 指令，将所有 `std` 名称整个地引入到当前空间：

```
// 例 40-2(c): 选择 C，使用 using 指令
//
#include <iostream>

int main()
{
    using namespace std; // 或者，放在文件范围
    cout << "hello, world" << endl;
}
```

要么，综合使用以上方法。

那么，从长远来看，使用名字空间的正确方式是什么？既想将大量的现有代码尽快移植到新的标准上，同时又想将（对现在来说）不必要的移植工作延迟到将来，并不对以后的移植工作增加任务量，最佳方式是什么？本条款的余下部分将回答这些问题。

好的长期方案的设计准则

首先，从长远来看，使用名字空间的正确方式是什么？先确定这一点很重要，因为只有知道了长期方案，在制定最佳短期移植策略时，我们才能确定自己的目标。

简而言之，在使用名字空间时，一个好的长期方案至少应该遵循以下规则：

名字空间使用规则 1：绝对不要在头文件中使用 using 指令

规则 1 的理由是，using 指令会恣意污染名字空间，因为它有可能引入大量名称，而其中的许多名称（通常绝大多数）是不必要的。不必要的名称一旦出现，名称冲突的可能性就会在无意中大大增加——不仅头文件本身会如此，每一个包含（#include）了这个头文件的模块也是如此。你可以将头文件中的 using 指令看作一群四处疯狂掠夺的野蛮人，它们对所过之处肆意破坏；仅仅只是出现它们的身影，“无意的冲突”就会发生——即使你认为自己已经和它们结成了盟友。

名字空间使用规则 2：绝对不要在头文件中使用名字空间 using 声明

（注意这里的限定词——“名字空间”using 声明。当然，你可以放心地写一个类成员的 using 声明，以引入所需的基类名称。）

这条规则可能会让你感到吃惊，因为和最为普遍的建议相比，它的要求又进了一步。大多数程序员的建议是，using 声明绝对不要出现在共享头文件的“文件范围”。就其本身而言，这是一条好建议；因为，和 using 指令一样，文件范围内的 using 声明也会导致同样的名字空间污染，只不过少一些而已。

依我来看，上面的“普遍建议”还不够。你根本就不应该在头文件中使用 `using` 声明，即使是在某个名字空间范围也不应该。`using` 声明的含义是会发生变化的，在某个模块中，它取决于哪些头文件被包含（`#include`）在这个 `using` 声明之前。这种不可预测性绝对应该禁止，因为这种做法不仅糟糕，而且还会导致程序不合法。考虑这样一种可能的情况：如果头文件中有某个函数 `f()` 的内联定义（例如，假设 `f()` 是一个模板，或是某个模板的一部分），而且，`f()` 中某行代码的含义会根据 `#include` 指令发生变化，这种情况下，这整个程序就是不合法的，因为它违反了“一个定义规则”（ODR，One Definition Rule）。ODR 要求，在每一个使用了某个实体（这里，也就是 `f()`）的编译单元中，这个实体的定义必须完全相同。如果不是这样，我们就会被流放到“具有不可预测行为的荒原”^②。即使在没有违反 ODR 的情况下，不可预测性也不是件好事；在后面的例 40-3(c) 中，我将再次说明这一问题。

名字空间使用规则 3：在实现文件中，绝对不要在`#include` 指令之前使用 `using` 声明或 `using` 指令。

所有的 `using` 声明和 `using` 指令必须跟在所有的 `#include` 指令之后。否则，由于 `using` 会引入意想不到的名称，（通过 `#include` 指令）被包含进来的头文件的语义就有可能发生变化。

除此之外，在实现文件中，你可以继续使用 `using` 声明和（或）`using` 指令的任意组合，只要有意义，无论是在文件范围和（或）函数范围，你都无需担心。在那个特定的文件中，你在局部范围作出的决定取决于怎样让自己觉得方便，并且不会对别的实现文件有潜在的副作用。`using` 声明和 `using` 指令的存在是为了给你提供方便，而不是要你为它们提供方便。做你想做的（如果没有名称冲突，使用所有名称又有什么错？），只要它影响的只是你自己的编译单元，而不是别人的头文件或编译单元。

“一定意义上的”名字空间使用规则 4：在使用 C 头文件时，采用新风格的`#include <chandler>`，而不采用旧风格的`#include <header.h>`。

在实践中，这条规则算不上什么大不了的事，但出于理论上的完整性，并且，为了不至于让信奉标准的人找我的麻烦，我还是提到这一点。为了向后和 C 兼容，C++ 还是支持所有的标准 C 头文件（例如，`stdio.h`），当你包含（`#include`）那些原始版本的头文件时，相应的 C 库函数会像以前一样现身于全局名字空间。但同时，C++ 标准也表示不赞成使用旧式头文件名，这意味着通告大家，C++ 标准的未来版本会删除它们。因而，标准 C++ 强烈鼓励程序员选择使用新版本的 C 头文件，这些头文件名称以“c”开头，

扩展名 “.h” 被去掉了（例如，`cstdio`）。在用新名称包含（`#include`）C 头文件时，你得到的是相同的 C 库函数，只不过，如今它们存在于名字空间 `std` 之中。

那么，为什么说它算不上什么大不了的事呢？首先，尽管官方正式反对“`name.h`”这样的名称，但你尽可以放宽心：这些名称绝不会真正消失。非常坦白地说——但只能天知地知你知我知：即使标准委员会将来什么时候真的删除了它们，标准库供应商还是会通过捆绑的方式提供那些旧式头文件。为什么？因为作为他们的客户，你有太多太多的现有产品代码在使用那些旧式头文件。而且，C++ 标准委员会当然也十分清楚这一点，因为委员会中的一部分人就来自于那些供货商。这一点已经说得够多了。第二点，有时候，出于某种原因你可能会选用“`name.h`”头文件，因为其它某些标准，例如 Posix，需要在“`name.h`”头文件中增加新的名称。如果你在使用那样的扩充功能，理论上来说，使用“`name.h`”风格的头文件会更安全——即使标准库供应商有可能会在“`cname`”头文件中也提供那些名称（即使标准没有正式要求标准库供应商这样做，但为了向你提供方便性和一致性，他们可能会将其作为普通的扩充功能提供给你。）

长期方案：一个启发式的例子

为了阐释前面的规则，先考虑下面的程序模块；然后，为了将这个程序模块移植到名字空间，我们来看看两个长期方案。

```
// 例 40-3(a): 没有名字空间的原始代码
//



//--- 文件 x.h ---
//
#include "y.h" // 定义 Y
#include <deque.h>
#include <iostfwd.h>

ostream& operator<<( ostream&, const Y& );
Y operator+( const Y&, int i );
int f( const deque<int>& );

//--- 文件 x.cpp ---
//
#include "x.h"
#include "z.h" // 定义 Z
#include <ostream.h>

ostream& operator<<( ostream& o, const Y& y )
{
    // ... 在实现中使用 z ...
    return o;
}

Y operator+( const Y& y, int i )
```

```

{
    // ... 在实现中使用另一个 operator+() ...
    return result;
}

int f( const deque<int>& d )
{
    // ...
}

```

要想将上面的代码迁徙到“支持名字空间的编译器”和“将标准名称都放在名字空间 std 之中的程序库”中，最好的途径是什么？继续阅读本条款之前，请暂停一下，想想你会找到哪些选择方案？哪些方案好？哪些不好？为什么？

已经作出选择了吗？好啊，那么继续吧。为了达到这一目标，你可能有好几种途径，我将介绍两个常见的策略——其中只有一个堪称上佳。

一个好的长期方案

这个好的长期方案是：在头文件（.h 文件）中，为每一个出现的标准名称都显式地加上修饰；在每一个源文件（.cpp 文件）中，为了方便，只用为所需的名称加上 using 声明——因为在那个源文件中，那些名称可能会被大量使用。

```

// 例 40-3(b)：一个好的长期方案
//

----- 文件 x.h -----
//
#include "y.h" // 定义 Y
#include <deque>
#include <iostream>

std::ostream& operator<<( std::ostream&, const Y& );
Y operator+( const Y&, int i );
int f( const std::deque<int>& );

----- 文件 x.cpp -----
//
#include "x.h"
#include "z.h" // 定义 Z
#include <iostream>
using std::deque; // “using”出现在所有的#include 之后
using std::ostream;
using std::operator+;
// 或者，如果合适的话，使用 “using namespace std;”

```

```

ostream& operator<<( ostream& o, const Y& y )
{
    // ... 在实现中使用 z ...
    return o;
}

Y operator+( const Y& y, int i )
{
    // ... 在实现中使用另一个 operator+() ...
    return result;
}

int f( const deque<int>& d )
{
    // ...
}

```

注意，x.cpp 中的 using 声明应该出现在所有#include 指令之后；否则，它们可能会给别的头文件带来名称冲突，具体情况取决于这些头文件被包含（#include）的次序。举个例子，使用 x.h 的 operator+() 可能会有歧义，这取决于是否有别的 operator+() 函数可见，而后者又取决于哪些标准头文件刚好被包含在 using 声明之前或之后。

我有时遇到这样一些人，即使在他们的.cpp 文件中，他们也完全放弃了“using”的使用，而是在每一个标准名称前都显式地加上修饰。我不推荐这样做，因为这需要大量的额外工作，但通常不会带来任何实际好处。

一个不那么好的长期方案

与此相反，另一个常被推荐使用的方案实际上很危险。这个糟糕的长期“方案”建议：将项目中所有的代码都置于自身的名字空间之中（这种做法本身没错，但它没有你所想象的那样有用），并在头文件中使用 using 声明或 using 指令（无意之中，它为潜在的问题敞开了大门）。之所以会有人建议使用这种方法，原因在于：和其它某些名字空间迁徙方案相比，在修改代码时，它需要的打字工作量最少：

```

// 例 40-3(c)：一个错误的长期方案（或者，
// 为什么绝对不能在头文件中使用 using
// 声明——即使是在一个名字空间中）
//

//--- 文件 x.h ---
//
#include "y.h" // 定义 MyProject::Y，并在名字
               // 空间 MyProject 中增加 using
               // 声明或指令

#include <deque>
#include <iostream>

```

```

namespace MyProject
{
    using std::deque;
    using std::ostream;
    // 或者，“using namespace std;”

    ostream& operator<<( ostream&, const Y& );
    Y operator+( const Y&, int i );
    int f( const deque<int>& );
}

//--- 文件 x.cpp ---
//
#include "x.h"
#include "z.h" // 定义 MyProject::z，并在名字
               // 空间 MyProject 中增加 using
               // 声明或指令
// 错误：z.h 中的 using 声明将来可能有名称上的
//        歧义。具体情况取决于：如果在某个模块
//        中，某些头文件被包含在 z.h 之前，那
//        么，那些头文件中存在哪些 using 声明
//        将会影响到 z.h 中 using 声明的含义。
//        （在本例中，x.h 或 y.h 可能会给
//        它带来意义上的改变）
#include <iostream>

namespace MyProject
{
    using std::operator+;

    ostream& operator<<( ostream& o, const Y& y )
    {
        // ... 在实现中使用 z ...
        return o;
    }

    Y operator+( const Y& y, int i )
    {
        // ... 在实现中使用另一个 operator+() ...
        return result;
    }

    int f( const deque<int>& d )
    {
        // ...
    }
}

```

注意那个醒目的错误。错误的原因正如上面所述：在一个头文件中，using 声明的含义可能会发生变化——即使 using 声明是存在于一个名字空间中，不是在文件范围——这具体取决于：在一个客户程序模块中，另外有哪些头文件被包含（#include）在它的前面。例如，using std::operator<<语句可能会表示完全不同的含义，using std::operator+；也会如此——具体情况取决于：到这个 using 声明为止，已经包含了哪些头文件（更不

必说，这个答案会随着标准库实现的不同而变化）。显然，不管你在编写何种代码，如果随着头文件被包含的顺序不同，代码的含义会偷偷发生变化，这总不是件好事。

一个有效的短期方案

我之所以花时间来说明理想的长期方案，并质疑了几个不好的做法，原因在于：知道了想要达到的最终目标，在选择更简单的短期方案时，就不致影响到长期方案。那么，我们现在就来解决这个短期问题：要想移植你现有的代码库，使之能处理名字空间 std，最有效的途径是什么？

说实在的，在我们的产品开发周期中，“将编译器升级到新的版本”只不过是众多任务中的一个。在任务列表中，“升级编译器并将代码移植到名字空间”可能不是最紧迫的一项，你还有其它大量工作需要完成，也还有许多产品功能需要增加。更可能的是，你还承受着产品开发期限的压力，所以，你应该采用最快捷的名字空间迁徙方案，它能让你完成任务，但又不影响到将来的安全性和可用性。如何最好地将（对目前来说）不必要的移植工作延迟到将来，同时不增加以后整个移植工作的任务量呢？首先，考虑头文件的移植。

移植步骤 1：在每个头文件中，为所有所需之处添加“std::”修饰符。

“真的有必要在每个地方添加 std:: 吗？”你会问。“难道不能只是在头文件中加上 using 声明或 using 指令吗？”是的，添加 using 声明或 using 指令所需要的工作量确实最小，但以后，当你正确实施长期方案的时候，这些工作都得重新撤消。在本条款前面提出的名字空间使用规则 1 和 2 中，我们已经否决了这种方法；我希望你能同意，我们已经见证了很有说服力的理由，它告诉我们不要走进那条死胡同。既然如此，我们决不能在头文件中添加 using 声明或 using 指令；在头文件中，我们唯一的选择是在合适的地方添加“std::”。

使用自动文本替换功能，你可以迅速完成大部分工作——例如，使用一个工具，在所有“*.h*”文件中将“string”修改为“std::string”（vector、list 等常见名称也如法炮制）。要注意的一点是，在你使用的其它名字空间中，这些常见名称可能也会出现，不要无意中把它们也替换掉了。例如，在你使用的某个第三方程序中，代码中有它自己的非标准的 string，对于这样的名称，你就只能手工处理。

移植步骤 2：创建一个名为 `myproject_last.h` 的新头文件，使之包含 `using namespace std;` 指令。然后，在每一个实现文件中，在所有其它 `#include` 指令之后包含（`#include`）`myproject_last.h`。

对实现文件来说，情况好办一些。我们只需在每一个实现文件中简单地写上一个 `using` 指令就可以万事大吉；当然，`using` 指令要出现在所有 `#include` 语句之后。这种做法不违反规则 1 的精神，因为这里的头文件很特殊。它不是一种普通头文件，用来声明某种东西然后供以后使用；相反，它是一种机制，用来将代码插入到实现文件中，而且，是插在实现文件中某个易于控制的特定之处。

较之在所有实现文件中写上“`using namespace std;`”，这样做有一点小小的好处。在很多工程中，有这样一个头文件是很常见的事；也就是说，你可能已经有了某个像 `myproject_last.h` 这样的头文件。如果是这样，你只用写上一行代码就可以大功告成。无论如何，在每个文件中写“`using namespace std;`”也好，或者在每个文件中写“`#include "myproject_last.h"`”也好，总的工作量是相同的。

最后一点，怎么处理新的`<header>`头文件风格呢？幸运的是，这是一个可选任务；所以，在初次迁徙名字空间的过程中，这一步不需要做（或者，实际上永远都不需要做）。

针对例 40-3(a)，在运用两步移植策略之后，结果如下例所示：

```
// 例 40-3(d)：好的短期方案，  
// 运用了两步移植策略  
  
// --- 文件 x.h ---  
//  
#include "y.h" // 定义 Y  
#include <deque>  
#include <iostream>  
  
std::ostream& operator<<( std::ostream&, const Y& );  
Y operator+( const Y&, int i );  
int f( const std::deque<int>& );  
  
// --- 文件 x.cpp ---  
//  
#include "x.h"  
#include "z.h" // 定义 Z  
#include <iostream>  
#include "myproject_last.h"  
// 在其它所有#include 之后
```

```

ostream& operator<<( ostream& o, const Y& y )
{
    // ... 在实现中使用 z ...
    return o;
}

Y operator+( const Y& y, int i )
{
    // ... 在实现中使用另一个 operator+() ...
    return result;
}

int f( const deque<int>& d )
{
    // ...
}

//--- 公共文件 myproject_last.h ---
//using namespace std;

```

这不会累及长期方案，因为它所做的任何事情都不需要长期方案去“撤消”。同时，和完整的长期方案相比，它更简单，需要的代码修改量更少。实际上，要想让代码在支持名字空间的编译器上工作，并且不至于以后还得回过头去撤消已做的工作，这种方法所需要的工作量最少。

结尾：转向长期方案

最后，很可能在以后某个适当的时候，你暂时不再为工程期限所迫，你就可以实施简单的移植策略，过渡到例 40-3(b)所介绍的完整的长期方案。简单地遵循以下步骤。

(1) 在 myproject_last.h 中，注释掉 using 指令。

(2) 重新编译你的工程。看看哪些地方不能通过编译。然后，在每个实现文件中，增加正确的 using 声明和(或)using 指令——要么在文件范围(位于所有#include 之后)，要么在每个函数内部，视具体情况而定。

(3) 这一步可做可不做。在每一个头文件或实现文件中，将“包含 (#include) C 头文件的代码”修改为新的<header>形式。例如，将#include <stdio.h> 修改为#include <cstdio>。这可以通过自动文本替换工具很快完成。还可以更快！因为我们甚至可以完全省略这一步。

如果采纳了这一建议，即使工程期限迫在眉睫，你也可以快捷高效地将你的代码库迁徙到支持名字空间的编译器和程序库中，同时，完全不会损及长期方案。

后记

我希望，对于本书及其前任 *Exceptional C++*，你感兴趣的不仅仅是其中的难题和疑问，还包括所提供的解决方案和设计技术。这些解决方案和设计技术大多数都经过精心设计，因而可以直接应用于你目前的产品开发中，或是为你的产品开发带来帮助。我希望，你已经在此发现了一些东西，并早已将它们应用到你的日常工作中。如果是这样，本书的目的就达到了。

和以前一样，我还得补充一句：本书的后续内容还有很多。目前，在互联网上，新的 Guru of the Week 议题继续在 comp.lang.c++.moderated 新闻组上发布、探讨和争论，并收集整理在官方网站 www.Gotw.ca 上。对于这些议题，以及其它更多内容，我会及时将其整理归纳到 *Exceptional C++* 丛书的第三卷，也就是最后一卷——*Exceptional C++ Style* 之中。在下一本书中你将看到很多内容，下面列出的只是其中的一部分。

- 针对前两本书中已经熟悉的话题，提供了更多更新的素材。例如：泛型程序设计和模板技术；通过封装技术设计健壮的类，对派生类施加控制的方法；效率与优化；高效地使用标准库，以及在目前常见的平台上，不同的标准容器实际（而非理论）使用内存和资源的数据分析；不可或缺的异常安全议题和技术，等等。这些素材扩充了本书很多条款的内容——如条款 1 至 8, 12 至 23, 26 至 28 等。
- 解决现实世界问题的实用知识。例如，如何编写易于调试的代码；高效的数据格式；压缩技术，等等。
- 有关内存管理议题的实用性讨论。在现实世界中的不同平台上，内存管理问题如何表现各异，如何潜在地影响你的代码——即使在这些平台上，你使用的是完全符合标准的编译器。

- 还有，最重要的内容：一组来自现实世界的案例分析。它剖析了一组实际产品的代码和书面发表的代码，就 *Exceptional C++* 丛书所推荐的卓越的设计准则作了具体的示范。

再次感谢所有关注并支持 *Exceptional C++* 和本书的朋友。我希望，你能发现这些内容对你的日常工作有用，能帮助你继续撰写更快、更整洁、更安全的 C++ 代码。

附录 A: (在多线程环境下)并非优化

本附录和条款 13 至 16 以及附录 B 密切相关。

介绍

目前你正在编写多线程程序吗？可能是；或者，你所在的开发小组中有其他人是。多线程越来越普及，因为它能使程序具有更好的响应性、更有效率、在多处理器计算机上的分布性更好。但即使你只是在编写单线程程序，请还是继续阅读本附录。如果你所在的团队（或共享程序库）中有其它工程（projects）是多线程的，你使用的某些共享库可能就是多线程安全（multithread-safe）的版本，那么，下面将要讨论的一些问题此时此刻可能也正在影响着你。

你关心性能和优化吗？同样，可能你关心。无论硬件速度有多快，性能总是很重要的。优化和性能经常相互联系，譬如，为了获得更高的性能，我们会使用多线程，让等待不同资源的操作并行执行。

大惊小怪为哪般？

本附录的要点在于：在单线程模式下，某些广泛使用的程序库优化技术的确很有效率；但是，如果在建立（build）这个程序库（译注：意即对程序库代码进行编译）时顾及到了多线程环境下使用的可能，这些优化实际上会降低性能。我所说的都是很常见、很普及的优化技术，你可能早就在使用它们，但连自己都不知道——这是因为，在很多很普及的商用程序库中，都深藏着这些优化。

还有更坏的消息：这种性能上的影响往往很严重，远远超过它应有的程度，即使是在目前最普及的一些商用程序库中，情况也是如此。为什么？这是因为，要使代码

做到线程安全，方法往往不止一个；因而，一些程序库的设计者会选择那些适用于所有场合的方法，而这些方法对于某些特定场合并非最有效率。即使有更高效的选择方案，某些很普及的程序库还是会选用通用结构，例如互斥体。这就带来了很大的不同。

下面是最糟糕的一条消息。前面我说过这样一句话：“如果在建立这个程序库时顾及到了多线程环境下使用的可能”。它的含义是：即使你的程序是单线程的，性能上的损失也会影响到你。

在本次讨论中，我们将再次看到一种优化，这种优化通常是一种错误的优化，但在商用程序库中却很常见。我还将演示一种检测性能指标的简单方法，这样，如果你自己使用的程序库中隐藏了这种类似的缺陷，你就可以检测自己是否受到它的影响。如果你的确受到了影响，你就可以收集这些资料，并把情况报告给你的程序库供应商，这样，他们才能纠正这些问题。

在阐述这一普遍问题时，我将采用一个具体的例子——即在条款 15 和 16 中设计的 Optimized::String 类，这个类使用了 copy-on-write (COW) 优化。但请注意：

- COW 优化不是唯一的肇事者。无论何种“对用户透明的优化”，只要它们在底层有两个或更多的对象共享信息或状态，而调用者代码无法知道其内幕，我在下面所讨论的内容就对其适用。这包括“被共享的实现”（shared-implementation）或 COW 这样的“缓式拷贝优化”（lazy-copy optimization），但不仅仅局限于它们。

- String 不是唯一的肇事者。有很多数据结构，其拷贝操作的开销很昂贵，例如集合和容器；上述优化也会（并且已经）应用在很多这样的数据结构上。

请注意，虽然我们的示例代码刚好是用 C++ 来编写的，但是，这里的讨论适用于用任何语言编写的代码。我之所以要写这样一个专题，一个原因是：在业内某些圈子中，C++ 的标准 string 有一个不光彩的名声，即，“又慢又缺乏效率”。是的，在目前市场上，确实存在一些无谓地缺乏效率的 C++ string 实现。但我想强调的是“无谓”两个字，并指出也还是有高品质的实现，同时向你展示如何去辨别它们，从而避免前者。

回顾：普通的旧式 Original::String (条款 13)

首先看看代码示例，这里使用的是一个“普通的旧式字符串”，就像条款 13 中的 Original::String 那样，没有在底层运用任何 copy-on-write (或其它共享信息) 优化。无论何时，只要你对这种字符串进行拷贝或赋值，就会立即执行深拷贝操作；所以，只要拷贝或赋值操作一结束，你就真的拥有了两个分离的、独立的字符串对象，它们在底层没有任何联系。

对于这样一个普通的旧式字符串，为了在单线程和多线程程序中安全地使用它们，调用者代码需要做些什么呢？

使用普通的旧式字符串：单线程

假设在某个程序中，我们想记录遇到的最后一条出错信息，并将出错信息的产生时间自动添加到信息中。我们可以用一个全局字符串值来实现这一出错信息，并提供辅助函数，用以获取和设置这一状态值：

```
// 例 A-1: 一个简单的错误记录子系统
//
String err;
int count = 0;

String GetError()
{
    return err;
}

String SetError( const String& msg )
{
    err = AsString( ++count ) + ": ";
    err += msg;
    err += " (" + TimeAsString() + ")";
    return err;
}
```

只要程序是单线程的，我们就不必担心 GetError() 或 SetError() 会在同一时刻被不同的线程调用；所以，一切都没问题。例如，假设有这样的代码：

```
String newerr = SetError( "A" );
newerr += " (set by A)";
cout << newerr << endl;

// ...稍后...
//
newerr = SetError( "B" );
newerr += " (set by B)";
cout << newerr << endl;
```

输出就会像下面这样：

```
1: A (12:01:09.65) (set by A)
2: B (12:01:09.125) (set by B)
```

使用普通的旧式字符串：多线程

也许你对线程安全议题已经很熟悉，而且，你也知道“为什么”以及“如何”通过互斥体来对“访问操作”进行串行化控制，从而在线程之间安全地共享资源；即便如此，请你无论如何还是浏览一下本节。对于后面更详细的例子来说，本节是基础。

一言以蔽之：如果程序是多线程的，情况会有些不同。GetError() 和 SetError() 这样

的函数很可能在同一时间被不同的线程调用。这种情况下，对这些函数的调用可以交叉执行 (interlaced)，它们就不会再像上面所写的那样正常运作。举例来说，如果下面两段代码会在同一时间执行，想想会发生些什么：

```
// 线程 A
String newerr = SetError( "A" );
newerr += " (set by A)";
cout << newerr << endl;

// 线程 B
String newerr = SetError( "B" );
newerr += " (set by B)";
cout << newerr << endl;
```

导致它出错的方式很多。例如：假设线程 A 正在运行，并且，在 SetError("A")函数调用的内部，目前 err 被设置为 “1:”；恰在此时，操作系统抢占 (preempt) 其执行权，切换到线程 B。线程 B 完全执行结束后，线程 A 重新获得执行权并运行至结束。这种情况下，输出将是：

```
2: B (12:01:09.125) (set by B)
2: B (12:01:09.125)A (12:01:09.195) (set by A)
```

还可以很容易地设想出其它一些情形，从而产生更奇怪的输出；^①但在现实中，如果能得到像上面那样还算合理的输出，你已经十分幸运了。因为，一个线程的执行权可以在任何时候被抢占，这包括一个 String 操作（例如 String::operator+=()）正执行到中间的时候；当不同线程上的 String 成员函数同时修改相同的 String 对象时，你更可能看到的是间歇性的程序崩溃。（如果你此时尚未明白问题所在，请试着写出 String 的那些成员函数，看看它们交叉执行时会发生什么。）

纠正这一错误的方法是：在任一时刻，确保只有一个线程可以对共享资源进行操作。为了防止函数交叉执行，我们利用互斥体或类似设备 (device) 对它们进行“串行化”控制。但谁应该负责这种串行化工作呢？我们可以在两个层次上做到这一点。需要权衡的地方主要在于，在越低的层次上执行这一工作，需要的锁定操作就越多，这往往没有必要。因为，对于某个操作，较低的层次不知道是否应该加锁，所以为了以防万一，它们不得不每次都加锁。过多的锁定操作是个大问题，因为在大多数系统上，获取一个互斥体锁 (mutex lock) 的操作往往很昂贵，其开销接近或超过一个通用的动态内存分配操作。

(1) [错误的方案] 在 String 成员函数中执行锁定操作。采用这种方法，String 类将负责其所有对象的线程安全性。选择这种方法是个坏主意，这基于两个 (可能很明显的) 原因。

^① 只要在 SetError() 调用和随后的 cout 语句之间中断线程，就可以影响输出的次序（但不是内容）。留给读者的习题：在标准 iostream 子系统和调用者代码内部，cout 本身有哪些线程安全方面的问题？首先考虑部分输出的次序问题 (partial output)。

首先，它解决不了问题，因为它是作用在错误的层次上。这种做法只是保证 String 对象不会被破坏(也就是说，就 String 本身而言，它们还是有效的 String 对象)，但对于 SetError() 函数的交叉执行，它无法控制 (String 对象最终拥有的可能还是出人意料的值，就像前面所演示的一样。也就是说，对错误记录模块而言，它们不是有效的错误信息)。其次，它会严重地降低性能，因为锁定操作将会很频繁，远远大于本该需要的次数——每一个修改 String 的操作至少需要一次锁定；甚至，非修改操作可能也是如此！

(2) [正确的方案] 在拥有/操作 String 对象的代码中执行锁定操作。这种选择总不会错。这不仅仅因为，锁定操作只是在真正需要的时候才会执行，而且因为，它是作用在正确的层次上。我们所锁定的“操作”不是在一个“低层” String 成员函数之中，而是在一个“高层” 错误记录模块的消息格式化函数之中。此外，在错误记录模块中，“对出错信息字符串进行串行化控制”的额外代码还被隔离出来。

(3) [有问题的方案] 以上两种方法同时运用。在这个例子中，到目前为止，方案(2)足以独当一面。方案(1)则明显是错误的，你会奇怪我为什么还要提到它。原因很简单：在稍后的讨论中我们将看到，copy-on-write “优化” 会强迫我们作出一个有损性能的选择，它要求在类的内部执行所有锁定操作。但因为（正如前面所说）通过方案(1)本身无法真正解决所有问题，所以，最终我们不会完全用方案(1)取代方案(2)，而是在使用方案(1)的同时再加上方案(2)。正如你所想象的那样，这的确是个不好的消息。

在我们的例子中，要实现方案(2)，错误记录子系统对它拥有的 String 对象应该负责串行访问控制（译注：确切来说，“串行访问控制”意指“对数据访问操作进行串行化控制”，为了表述上的方便，故简称之，后同）。下面是一种典型的做法：

```
// 例 A-2: 一个线程安全的错误记录子系统
//
String err;
int count = 0;
Mutex m; // 保护 err 值和 count 值

String GetError()
{
    Lock<Mutex> l(m); // --进入互斥代码块-----
    String ret = err;
    l.Unlock();          // --退出互斥代码块-----
    return ret;
}

String SetError( const String& msg )
{
    Lock<Mutex> l(m); // --进入互斥代码块-----
    err = AsString( ++count ) + ":" ;
    err += msg;
    err += " (" + TimeAsString() + ")";
    String ret = err;
    l.Unlock();          // --退出互斥代码块-----
    return ret;
}
```

一切都不错，因为对 err 和 count 来说，每个函数体如今都是不可分割的原子 (atomic) 操作，因而不会发生交叉执行的情况。SetError() 调用被自动地串行化，它们的函数体绝对不会交叉执行。

有关线程、关键段、互斥体、信号量、竞态条件、哲学家进餐问题，以及大量有趣的相关议题，请参阅任何一本有关操作系统或多线程编程的优秀教材。在下面的讨论中，我假设你已经了解了这些基础知识。

在例 A-2 以及后面演示的类似代码中，我们使用了一个辅助的锁管理器类 (helper lock manager class)。利用这个辅助类，我们确保可以迅速地得到一个锁，并且，在随后释放这个锁时，不多不少正好释放一次。一旦将这些操作信息包装在管理器类中，例 A-2 这样的代码就更具异常安全性；如果 String 操作碰巧抛出了异常，就不会留下一个“得到后永远没被释放”的锁。

辅助类 Lock 看起来像下面这样：

```
template<typename T>
class Lock
{
public:
    Lock( T& t )
        : t_(t)
        , locked_(true)
    {
        t_.Lock();
    }

    ~Lock()
    {
        Unlock();
    }

    void Unlock()
    {
        if( locked_ )
        {
            t_.Unlock();
            locked_ = false;
        }
    }
private:
    T& t_;
    bool locked_;
};
```

引入“优化”：Copy-On-Write(COW)

COW（写入时拷贝）可能是最有名的字符串优化技术，它是缓式拷贝技术的一种形式，其字符串表示体（译注：representation，在后文中，视上下文亦有可能译作“实体”）使用了引用计数。条款 14 至 16 介绍并阐释了这一优化技术。

COW 不只适用于字符串，但其用途十分简单直接。在对一个普通旧式字符串进行拷贝或赋值时，一般会涉及到一个通用的动态内存分配操作，而这一操作的开销往往十分昂贵（例如，和函数调用相比）；因而，拷贝一个字符串自然也就成为了一种昂贵的操作。与此同时，调用者代码会经常对字符串对象进行拷贝，但随后永远不会再对拷贝作修改。例如，在对一个字符串对象进行拷贝后，调用者代码对其可能只是执行某些只读操作，譬如将它打印出来或者计算它的长度，然后摧毁这个拷贝。果真如此的话，创建一个独立的拷贝似乎是一种浪费，因为到后来这个拷贝会发现，自己的存在实际上没有必要。

所以，COW 字符串（例如条款 14 至 16 中的 `Optimized::String`）的不同之处在于，它的拷贝构造函数最初只是执行一个浅拷贝操作，深拷贝则被尽可能地推迟。毕竟，在对任一（源或目标）字符串执行只读操作时，我们不在意是浅拷贝还是深拷贝。只是在对其中某个字符串进行修改操作时，这个字符串才不得不最终执行深拷贝。^②

COW 有用吗？在单线程程序中，有时有用。至于它是否真的是一种优化，这极大程度地取决于字符串将被如何使用。请阅读 Rob Murray 在[Murray93]中的讨论和测试结果，它提供了 COW 在现实世界中（但在单线程环境下）的更多信息。

当你转移到某个（有可能是）多线程的环境中时，有趣的事就真正开始发生了。你会看到，如果线程安全很重要，使用 COW 会招致意想不到的性能损失；而且，在某些很普及（但无谓地缺乏效率）的商用实现中，这种损失表现得十分严重。

使用 COW 字符串：多线程

现在，让我们回过头来看看我们的老朋友——错误记录子系统。下面给出的还是那段有问题的调用代码：

```
// 线程 A
String newerr = SetError( "A" );
newerr += " (set by A)";
cout << newerr << endl;

// 线程 B
String newerr = SetError( "B" );
```

^② 可以扩充 COW，使之支持子字符串共享，而不只是整个字符串。（参见[Niec00].）

```
newerr += " (set by B)";
cout << newerr << endl;
```

和前面一样，这里的问题是：为避免破坏“错误记录模块”的内部字符串，并为了产生合理的输出，我们必须提供某种串行化操作（例如，使用互斥体锁）。但是，谁应该负责这种串行化工作呢？再次看看那两个层次：

1. 在 Optimized::String 成员函数中执行锁定操作

和前面一样，这个方案还是存在两个问题。它是在错误的层次上解决 SetError() 函数的交叉执行问题；它会严重地降低性能，因为每一个修改 String 的操作都会至少执行一次锁定（甚至，非修改操作可能也会如此）。只不过，这里和前面有一个不同：这一次，方案 1 无论如何都是必需的（参见下文）。

2. 在拥有/操作 Optimized::String 对象的代码中执行锁定操作

再一次，这是必需的；否则，我们无法解决 SetError() 的交叉执行问题。

可惜的是，如今，前面两种方案都无法独立解决问题。我们需要是某种重量级的手段：

3. [必需的方案]以上两种方法同时运用

这一次，故事没那么美：方案 2 是必需的，但我们还必须运用方案 1；因为，拥有/操作 copy-on-write optimized::String 的代码如今无法独立完成使命。调用者代码自身无法以正确的方式锁定正确的字符串，因为它无法知道：在某一时刻，哪个可见字符串对象正在共享某个实体。让我们再次看看例 A-2 中那个线程安全的错误记录子系统。它执行的锁定操作还是必需的，但如今已经不够；因为，即使我们对内部 err 字符串对象执行了串行访问控制，但调用者代码的其它线程会获取和操作那个对象的拷贝，而这些拷贝可能正在共享实体。

例如，在线程 A 的 SetError() 调用中，所有操作都已被安全地串行化，假设它正在对错误记录模块的 err 字符串进行操作，与此同时，线程 B（被安全地串行化了的 SetError() 已经执行结束）正在向它的 newerr 字符串添加字符。请注意，线程 B 的 newerr 字符串当初是作为 err 字符串的拷贝而创建的。^③没有任何明显的东西将内部 err 字符串和线程 B 的外部 newerr 字符串联结在一起。实际上，错误记录子系统甚至无法知道有任何 newerr 字符串存在，反之亦然。但如果这两个字符串刚好都在共享实体（在本例中，这不是有可能，而是很可能），两个线程就会同时在同一实体上执行 String 的成员函数——即使在 String 的外部，一切都已被安全地串行化。^④

^③ 或者，如果编译器没有执行返回值优化，它将是“err 字符串的拷贝”的拷贝。这没有本质的不同。

^④ 某些尚在供应的很普及的商用库也存在不同形式的 COW 缺陷；在这些库中，修改一个可见的字符串会同时错误地修改另一个不同的可见字符串，原因在于没有及时执行深拷贝。

可见，这不足以让错误记录子系统履行其正常的安全职责。作为 COW 字符串，Optimized::String 对象还必须保护它们自身。这就必然给字符串类带来某些开销。问题是，如果字符串实现招致不必要的效率低下，开销将会极其昂贵。

问题有多严重？一位程序员最近提供了一份报告，他通过一个单线程测试程序，对他的供货商所提供的“基于 COW 的程序库”的特性进行了测试。在以多线程为使用目标而对程序库进行编译的情况下，测试程序的执行时间为 18 到 20 秒之间；若以单线程为目标编译程序库，执行时间在 0.25 秒之内。（实际差异要大于这些数据显示，因为在两种情况下，测试的时间值都包含程序启动时间。）请注意到这里很重要一点，这个程序员的测试程序是单线程的，并非多线程，但由于在编译程序库时是以多线程为使用目标，问题还是影响到了他。如果 20 个程序共享一个程序库，其中只有一个程序是多线程的，这个程序库在编译时就还是得以多线程为使用目标——那么，任何问题都会影响到所有 20 个程序。

轻微效率低下与严重效率低下

看来，线程安全的 COW 实现总是会带来串行化的开销；但是，在“必需的”开销和“无谓的”开销之间，区别还是很大。

长话短说（实际上，这段“话”非常长），线程安全的 COW 可以采用这样的方式来实现，即，只对引用计数访问进行串行化控制。（关于这一点的详尽说明，还有 COW 其它方面的详尽说明，以及其它线程安全实现的介绍，请阅读条款 14 至 16。此外，附录 B 提供了测试程序的下载链接，那是经过了优化的示例代码，针对一个“功能有一定限制的 String 类”，它所测试的实现多达七个——五个 COW 实现，两个非 COW 实现。这里的讨论是其中一些分析结果的总结和深化）

所以，这就是问题所在：需要进行串行化控制的共享数据只有唯一一个，即引用计数（通常是一个整数），所以，较之使用一个重量级的通用线程安全方案，你可以做得更有效率。在必须对较大的结构进行串行访问的时候，互斥体责无旁贷；但如果在你的系统中有如下原子整数操作，你就可以放弃互斥体而使用这些原子整数操作，即：原子递增（atomic increment）、原子递减与检测（atomic decrement-and-test）、原子比较（atomic-comparison）。这些原子整数操作还是会带来开销，因为它们还是慢于普通的整数操作；但和互斥体这种更复杂的结构相比，它们还是要快得多。它们往往可以在单条普通的汇编指令中实现，或是在很短的一组指令中实现。

为了对这种差异有些感性认识，我们来看看，在一个基于 C++ 的 String 类中，一个成员函数的各种实现——这个成员函数是 operator[]() 的非 const 版本，它返回一个引用，指向字符串中“位于某一偏移位置的字符”。^⑤对于一个普通（非 COW）实现来说，其

^⑤ COW 的支持者可能会抱怨说，对于有可能执行修改操作的函数如 operator[]() 的非 const 版本来说，COW 和非 COW 之间的区别是最明显的。这没错，但为了提供一个短小干净的例子来演示非 COW 和各种形式的 COW 在实现上的区别，这个函数最简单。

代码看起来会像下面那样，其中，buf_是一个指针，指向 String 对象的内部缓冲区，这个缓冲区存储的是 String 对象的实体：

```
// 普通的非 COW operator[] (非 const 版本)
//
char& String::operator[]( size_t n )
{
    return buf_[n];
}
```

COW 实现看起来也差不多，只是，它首先得保证实体不被共享（就 operator[]() 而言，函数必须返回一个指向非 const 对象的引用，而且实体必须标记为非共享，关于这两点的原因，我不想在此展开讨论，因为它们和这里的主题没有直接关系；更详细的介绍请参阅条款 16）：

```
// COW operator[] (非 const 版本)
//
const char& String::operator[]( size_t n )
{
    EnsureUnique();
    return data_->buf[n];
}
```

这里的关键在于，对于不同形式的 COW，EnsureUnique() 的内部应该怎样实现。下面是几个简化了的实现，它们省略了“非共享”标志和其它更复杂的操作。这里的 data_ 指向的是内部（可能被共享的）实体，此实体包含引用计数值（refs）：

```
// 非线程安全的 COW: 没有锁定机制
//
void String::EnsureUnique() const
{
    if( data_->refs > 1 )
    {
        StringBuf* newdata = new StringBuf( *data_ );
        --data_->refs; // 现在，所有实际工作已经完成，
        data_ = newdata; // 所以，获得拥有权
    }
}

// 线程安全的 COW: 原子整数操作
//
// 使用原子整数调用来串行访问 data_->refs。
// 注意，IntAtomic* 调用不一定是函数调用，
// 它们可以像纯粹的汇编指令那样高效。当然，
// 它们还是会带来开销，因为 EnsureUnique
// 必然会被每一个“可能有修改操作”的 String
// 函数来调用，而 IntAtomic* 操作比普通的
// 整数操作要慢。
//
```

```

void String::EnsureUnique() const
{
    if( IntAtomicCompare( data_->refs, 1 ) > 0 )
    {
        StringBuf* newdata = new StringBuf( *data_ );
        if(IntAtomicDecrement( data_->refs ) < 1 )
        {
            delete newdata;      // 防止两个线程同时
            data_->refs = 1;     // 执行此操作
        }
        else
        {
            // 现在, 所有实际工作已经完成,
            data_ = newdata;     // 所以, 获得拥有权
        }
    }
}

// 线程安全的 COW: 互斥体
//
// 每个 data_ 缓冲区都包含一个互斥体对象,
// 用以串行访问 data_->refs.
//
// 这一方法会造成不必要的效率低下, 但它还是
// 被应用在某些很普及的商用 string 库中.
// EnsureUnique 还是会被每一个“可能有修改
// 操作”的 String 函数所调用, 但较之使用
// IntAtomic* 函数的那个版本, 其开销要大.
//
void String::EnsureUnique() const
{
    Lock<Mutex> l(data_->m); //-----
    if( data_->refs > 1 )
    {
        StringBuf* newdata = new StringBuf( *data_ );
        --data_->refs;
        data_ = newdata;
    }
    l.Unlock(); //-----
}

```

COW 总是会增加开销——特别是, 如果我们要求它具有线程安全。但这也说明了一个道理, 即, 在原子整数操作可以单独完成任务的情况下, 我们没有理由去使用像互斥体这样的东西, 因为后者会带来无谓的效率低下。

一些实际测试数据

“必需的”开销和“无谓的”开销之间, 其差异很大。这里的例子源自条款 16 和附录 B 的测试结果(如果想在自己的系统上进行测试, 你可以下载整个测试程序的源代码, 下载链接提供在本书网页 <http://www.gotw.ca/publications/mxc++.htm> 上)。在这个

测试中，对一个长度为 50 的字符串，我在一个密集循环中对其不断执行拷贝操作。其中，三分之一的拷贝完全不被修改（这是 COW 优化的典型应用场合；在这种情况下，深拷贝得以完全避免），其余每个字符串拷贝不多不少被修改一次。我对六种形式的 String 类提供了测试结果：

- (1) “Plain” 是普通的非 COW 字符串。
- (2) “Plain_FastAlloc” 和 Plain 相同，但使用的是优化了的内存分配程序，而不是缺省的 operator new() 函数（但请注意，在我测试的编译器平台上，就其本身而言，后者的表现非常不错。）。
- (3) “COW_Unsafe” 是很有效率但非线程安全的 COW 字符串。
- (4) “COW_AtomicInt” 是很有效率且线程安全的 COW 字符串，它使用原子整数操作来串行访问引用计数。
- (5) “COW_CritSec” 和 AtomicInt 相同，但使用的是 Win32 关键段锁定机制，而不是原子整数操作。
- (6) “COW_Mutex” 和 AtomicInt 相同，但使用的是 Win32 互斥体锁定机制，而不是原子整数操作。

以下是测试结果说明，它基于我手边刚好有的一个编译器和 Windows 平台。请注意，在其它编译器和平台上，结果会有不同；请阅读附录 B，那里提供了更多的测试数据，以及一个源代码下载连接；下载了源代码，你就可以在自己的平台上进行编译和测试。

String 实现策略	耗时 (数字越小越好)
Plain	1530ms
Plain_FastAlloc	390ms
COW_Unsafe	1380ms
COW_AtomicInt	1750ms
COW_AtomicInt2	1490ms
COW_CritSec	7800ms
COW_Mutex	23010ms

当然，这反映的只是一种情况。它可能就是你的程序使用字符串的典型情况，也可能不是。我之所以扼要地展示这些数据，主要目的是想说明以下几点：

- 线程安全的 COW 必然比非线程安全的 COW 缺乏效率。在我们的测试中，线程安全开销所带来的差异使得我们的测试实际上变成了“COW 是一种优化还是一种恶化”的测试。
- 线程安全的 COW 如果实现得不好（特别是 COW_CritSec，它的实现方式和某种很普及的 string 库相同），就会招致严重的性能损失。注意：如果你的平台不支持原

子整数操作，你就只有别无选择地使用更笨重的方案。那么，几乎在所有场合下，COW 都会变成一种严重的恶化行为。

- 渴望优化，这本身没有错；但要想得到更好（而且更简单）的优化，应该在内存分配上动心思，而不是深拷贝。还请注意，优化内存分配程序对线程安全没有影响。准则：不要猜测，请实际动手测试！只有在测试出程序瓶颈的真正所在之后，你才能去优化。（当然，你总可以同时使用这两种优化，但开发时间绝不是无限的。想一想，哪种方案会让你收益更多？）

- 再次指出，测试程序是单线程的。无论你是否需要多线程，只要你的程序库在编译时是以多线程为使用目标，你就要承担它所带来的开销。

正如条款 16 所展示的那样，还有其它很多有趣的测试结果。例如，在一个具有高效的缺省内存分配器的编译器上，如果测试较大的字符串，我们会发现：COW 的优点并不表现在“避免了内存分配的成本”，而在于避免了“在字符串中拷贝字符的成本”——这可能是一个让你感到吃惊的结果。

一个现实世界中的例子：C++的 `basic_string`

你可能想知道，这些错误的优化是不是也影响到你。很大一种可能是，它们确实影响到你。一个常见的例子是，如果你在使用某种具有字符串类的面向对象语言，你就很有可能受到了它的影响。

纵观历史，长期以来，COW 一直都是种很普及的优化技术，在很多程序库中，它被应用于各种数据结构，包括字符串。由于这一历史原因，C++标准特意指出，对于标准 C++ 的 `basic_string` 模板，供货商在提供符合标准的实现时，可以选择使用 COW。结果也许不会让人感到意外，我所知道的所有早期实现确实都是这样做的，即使 COW 会给自身带来某些开销、^⑥更难以实现、而且常常会引发难以察觉的错误和影响。^⑦

几年前，我们小组引进了某个供货商的标准 C++ 程序库实现，并在单线程环境下对其进行了测试。然后，我们打开了多线程编译开关，对这个程序库进行了重新建立。结果发现，我们的很多程序比以前慢了好几倍。“出什么问题了？”我们不禁抓着脑袋，大惑不解。“测试程序应该没问题。我们只是在自己的代码中做了些修改，问题究竟在哪儿？是某些开销昂贵的互斥体影响到我们了吗？或是在其它什么地方导致了效率低下？”

借助分析工具，我们做了一些研究并找到了问题，但这却进一步加深了谜底。我们的程序耗费了大量的时间去执行加锁和解锁操作，而且这些操作的数量超出了正常的想象，

^⑥ 例如拥有权标志和引用计数。

^⑦ 要写出完善的 COW 代码，其复杂性超过你的想象。我所认识的最有经验的程序库设计者曾经再三告诉我，他们的 COW 代码不断地出现错误。要知道，这些人有数十年编写高质量程序库的经验，而不是新手。

远远大于我们自己的代码中应有的锁定操作次数。那么，所有这些锁定操作从何而来？

带着好奇，我们做了进一步的分析并最终让问题水落石出。在对程序库代码进行几分钟的分析之后，我们发现，和其他很多供货商一样，这个程序库供货商在它的 `basic_string` 实现中也使用了 COW 优化。很自然，我们广泛使用的是标准 C++ 字符串，而不是 C 风格的 `char*`，这样，几乎每一个字符串操作都导致加锁和解锁操作。

我们又花了几分钟，很快写出了一个只有 10 行代码的示例程序：在这个程序的一个计时循环中，我们设置了某些很合理的字符串操作。结果显示，在程序库被编译为多线程版本的情况下，较之程序库被编译为单线程版本，示例程序的执行速度慢了 34 倍。我们报告过这个问题，但就我所知，这个程序库目前还在使用 COW——而且是无谓地效率低下的 COW 实现。这个供货商的名字无关紧要，因为直至目前，在 Windows 以及其他平台上，所有最普及的 `basic_string` 实现都使用了 COW。并非所有基于 COW 的实现都这样效率低下，但有一些的确如此。

好消息是，在我写作本书之时，也就是 2001 年，我终于可以告诉你，整个业界似乎正在放弃 `basic_string` 的 COW 实现。在较新版本的编译器所提供的 C++ 标准库中，包含的只有非 copy-on-write string；这种趋势正在增长。这是条好消息，因为作为程序员，我们日益需要编写多线程应用程序，因而注意到了那些问题，并写文章就其进行讨论，最终，这些问题终于引起了供货商的注意。为我们提供更快的工具，这符合供货商的利益；这里，“快”的意思是：“在我们的使用方式下”快；时至今日，也就是：“在多线程环境下”快。

坏消息是，可惜，不是所有的供货商都已经转移到非 COW string。所以，你下一个问题自然会是：

“我受到影响了吗？”

如果你的程序库在建立时考虑到了多线程环境下使用的可能性，并且性能对你很重要，那么，你可以像我们前面所做的那样行事——而且，你也应该这样做。也就是：用分析工具对你的代码进行测试，找出你使用得最多的数据结构。你可以去分析成员函数的调用次数，而不一定是函数的执行时间。写一些测试程序，在循环中对那些数据结构进行操作；然后，针对程序库的单线程和多线程版本，分别计算那些程序的运行时间。如果二者的差异非常大，特别是，如果你能从差异大小看出它是否比应有的效率要低，那么，请将测试情况报告给你的供货商。

程序库的实现者一般都是通情达理的人。当然！我所遇见的都是这样。为客户提供高品质的代码，这是他们的职责。如果能让他们获得比其他程序库供货商更具竞争力的优势，他们通常乐于听从我们的建议，并提供我们想要的东西。在大多数场合下，“抛

弃 COW 实现”属于一种“好的优化”（通用程序库最关心的就是“大多数场合”）。

那么，COW 及其类似的“优化”为什么曾经风行一时呢？最大的原因可能是岁月的魔力。作为传统，COW 数十年来一直是很常用的优化。即使在 20 世纪末，尽管大量的商业应用急需多线程代码，但在业界到底有多少人在写多线程的商业程序，我们心中完全没有数。如今，情况正在发生改变。程序库供货商越来越清楚地看到，很大一部分客户正在使用程序库的多线程版本；所以，对那些有可能是错误的优化技术，如 copy-on-write，一些大的供货商已经决定弃之不用。对我以及很多人来说，这是件大好事。希望其他供货商也能够尽快跟进。

在多线程程序中，COW 招致的性能惩罚可能是惊人的。但请记住，问题不只是对 COW 而言。任何共享信息的优化，只要它们“在底层”是以某种方式将两个对象联系起来的，就都会带来同样的问题。

总结

对复杂的数据结构进行优化，这完全没有错。问题是，一些很常用的优化方法实际上并非总是优化，它们会带来性能上的惩罚；随着越来越多的程序员开始编写多线程程序，或是和其他编写多线程程序的人共享程序库，这些惩罚就会越来越引人注目——特别是，如果对线程安全问题缺乏经验，写出的程序库会无谓地缺乏效率，这时候，问题会更严重。

好消息是：采用其它优化方法可以提供相同的效果，却不会在多线程环境中损及性能。例如，在 C++ 中，我们可以定制 operator new()，从而提供一个优化的内存分配器，这通常是处理“内存分配”性能问题的好方法。特别是，对字符串来说，小字符串优化（在 String 对象中存储小字符串，而不是在单独的缓冲区中）总会带来稳定的性能；如果字符串长度在一定限度之下，内存分配会完全避免。

诀窍在于：使用真正的优化，而不是错误的优化；无意之中，后者对程序库代码的优化往往只针对于使用单线程的场合。

附录 B: 单线程 String 实现与多线程安全 String 实现的对比测试结果

本附录提供的是条款 16 和附录 A 的详细补充材料。如果想在自己的系统上进行测试，你可以在互联网上下载测试程序的源代码；下载链接提供在本书网页 <http://www.gotw.ca/publications/mxc++.htm> 上。

测试方法

为了评估 COW，我针对三种功能进行了测试：

- (1) 拷贝操作（COW 的出众之处，也是它存在的理由所在）。
- (2) 会引发内存重新分配的修改操作（此操作以 Append() 为代表，它会逐渐增加字符串的长度；测试这一操作是为了保证：我们作出的所有结论都考虑到了“正常使用字符串时所带来的周期性的内存重新分配”的开销）。
- (3) 可能会修改字符串的操作，字符串长度的改变不足以引发内存重新分配，或实际上根本不修改字符串（此操作由 operator[]() 担当）。

结果显示，在后两种情况下，每个操作所带来的开销都是常数（误差大约在 20% 之内），因而这两种情况大致可以放在一起考虑。为了简化问题，对那些会修改字符串并增加字符串长度的操作，如 Append()（235ms 的开销），以及那些有可能修改字符串的操作，如 operator[]()（280ms 的开销），我们假设它们被调用的频度大致相同，那么，在这个实现中，对于“修改操作”和“可能的修改操作”，COW_AtomicInt 的开销大约是每 1,000,000 个操作耗时 260ms。

最后，分别针对条款 16 中 2(a) 和 2(b) 的示例，我首先利用“原始测试值”一节中的数据，粗略地手工计算出预测性能值；然后，我再运行测试程序，得到实际性能值。

例 2 (a) 总结

预测值

COW_AtomicInt 的开销	Plain 的开销
1M 浅拷贝 及析构函数	1M 深拷贝 及析构函数
667K 修改操作	1600
667K 深拷贝	???
667K 深拷贝上的 额外开销	1060
667K 修改操作上的 额外开销	??? 175
	1635+
	1600+

测试值

(测试程序在一个密集循环中执行拷贝操作。其中，
33% 的拷贝通过 Append 被修改 1 次，
另外 33% 的拷贝通过 operator[] 被修改 1 次)

执行 1,000,000 次循环操作，字符串长度为 50:

Plain_FastAlloc	642ms	拷贝:	1000000	分配:	1000007
Plain	1726ms	拷贝:	1000000	分配:	1000007
COW_Unsafe	1629ms	拷贝:	1000000	分配:	666682
COW_AtomicInt	1949ms	拷贝:	1000000	分配:	666682
COW_AtomicInt2	1925ms	拷贝:	1000000	分配:	666683
COW_CritSec	10660ms	拷贝:	1000000	分配:	666682
COW_Mutex	33298ms	拷贝:	1000000	分配:	666682

例 2 (b) 总结

预测值

COW_AtomicInt 的开销	Plain 的开销
1M 浅拷贝 及析构函数	1M 深拷贝 及析构函数
1.5M 修改操作	400
500K 深拷贝	1600
500K 深拷贝上的 额外开销	800
1.5M 修改操作上的 额外开销	???
	390
	1590+
	1600+

测试值

(测试程序在一个密集循环中执行拷贝操作。其中，
25% 的拷贝通过 Append 被修改 3 次，
另外 25% 的拷贝通过 operator[] 被修改 3 次)

执行 1,000,000 次循环操作，字符串长度为 50:

Plain_FastAlloc	683ms	拷贝:	1000000	分配:	1000007
Plain	1735ms	拷贝:	1000000	分配:	1000007
COW_Unsafe	1407ms	拷贝:	1000000	分配:	500007
COW_AtomicInt	1838ms	拷贝:	1000000	分配:	500007
COW_AtomicInt2	1700ms	拷贝:	1000000	分配:	500008
COW_CritSec	8507ms	拷贝:	1000000	分配:	500007
COW_Mutex	31911ms	拷贝:	1000000	分配:	500007

原始测试值

测试 const 拷贝和析构: COW 的目标使用场合

说明:

- 在创建和销毁 const 拷贝时, COW_AtomicInt 总是比“普通的非线程安全 COW”的耗时多两倍以上。
- 如果一个字符串的拷贝后来被修改, COW_AtomicInt 会无可挽回地增加常数级开销(每 1,000,000 次 400ms), 这不包括在其它操作上的开销。
- COW 为小字符串所带来的主要好处大多可以无需 COW 而得到——即, 可以通过使用更有效率的分配程序、或采用小字符串优化(关于小字符串优化的更多介绍, 请参见附录 A) 来得到。当然, 你也可以同时运用多种优化技术——在使用 COW 的同时, 还使用高效的分配程序和(或) 小字符串优化。
- COW 为大字符串所带来的主要好处并不在于避免了内存分配, 而在于避免了字符拷贝(char copying)。

执行 1,000,000 次重复操作, 字符串长度为 10:

Plain_FastAlloc	495ms	拷贝:	1000000	分配:	1000003
Plain	1368ms	拷贝:	1000000	分配:	1000003
COW_Unsafe	160ms	拷贝:	1000000	分配:	3
COW_AtomicInt	393ms	拷贝:	1000000	分配:	3
COW_AtomicInt2	433ms	拷贝:	1000000	分配:	4
COW_CritSec	428ms	拷贝:	1000000	分配:	3
COW_Mutex	14369ms	拷贝:	1000000	分配:	3

执行 1,000,000 次重复操作, 字符串长度为 50:

Plain_FastAlloc	558ms	拷贝:	1000000	分配:	1000007
Plain	1598ms	拷贝:	1000000	分配:	1000007
COW_Unsafe	165ms	拷贝:	1000000	分配:	7
COW_AtomicInt	394ms	拷贝:	1000000	分配:	7
COW_AtomicInt2	412ms	拷贝:	1000000	分配:	8
COW_CritSec	433ms	拷贝:	1000000	分配:	7
COW_Mutex	14130ms	拷贝:	1000000	分配:	7

执行 1,000,000 次重复操作，字符串长度为 100:

Plain_FastAlloc	708ms	拷贝:	1000000	分配:	1000008
Plain	1884ms	拷贝:	1000000	分配:	1000008
COW_Unsafe	171ms	拷贝:	1000000	分配:	8
COW_AtomicInt	391ms	拷贝:	1000000	分配:	8
COW_AtomicInt2	412ms	拷贝:	1000000	分配:	9
COW_CritSec	439ms	拷贝:	1000000	分配:	8
COW_Mutex	14129ms	拷贝:	1000000	分配:	8

执行 1,000,000 次重复操作，字符串长度为 250:

Plain_FastAlloc	1164ms	拷贝:	1000000	分配:	1000011
Plain	5721ms	拷贝:	1000000	分配:	1000011[*]
COW_Unsafe	176ms	拷贝:	1000000	分配:	11
COW_AtomicInt	393ms	拷贝:	1000000	分配:	11
COW_AtomicInt2	419ms	拷贝:	1000000	分配:	12
COW_CritSec	443ms	拷贝:	1000000	分配:	11
COW_Mutex	14118ms	拷贝:	1000000	分配:	11

执行 1,000,000 次重复操作，字符串长度为 1,000:

Plain_FastAlloc	2865ms	拷贝:	1000000	分配:	1000014
Plain	4945ms	拷贝:	1000000	分配:	1000014
COW_Unsafe	173ms	拷贝:	1000000	分配:	14
COW_AtomicInt	390ms	拷贝:	1000000	分配:	14
COW_AtomicInt2	415ms	拷贝:	1000000	分配:	15
COW_CritSec	439ms	拷贝:	1000000	分配:	14
COW_Mutex	14059ms	拷贝:	1000000	分配:	14

执行 1,000,000 次重复操作，字符串长度为 2,500:

Plain_FastAlloc	6244ms	拷贝:	1000000	分配:	1000016
Plain	8343ms	拷贝:	1000000	分配:	1000016
COW_Unsafe	174ms	拷贝:	1000000	分配:	16
COW_AtomicInt	397ms	拷贝:	1000000	分配:	16
COW_AtomicInt2	413ms	拷贝:	1000000	分配:	17
COW_CritSec	446ms	拷贝:	1000000	分配:	16
COW_Mutex	14070ms	拷贝:	1000000	分配:	16

测试 Append0: 一个始终执行修改，并周期性执行重新分配的操作

说明:

- Plain 总是比 COW 表现优异。
- COW_AtomicInt 和 Plain 在开销上的差异不大受字符串长度的影响，其差值大约是个常数，即，每 1,000,000 次操作相差约 235ms。
- COW_AtomicInt 和 COW_Unsafe 在开销上的差异不大受字符串长度的影响。其差值大约是个常数，即，每 1,000,000 次操作相差约 110ms。
- 对于较长的字符串来说，更佳的总体性能是由内存分配策略（见条款 13）带来的，和 COW 或 Plain 无关。

执行 1,000,000 次重复操作，字符串长度为 10:

Plain_FastAlloc	302ms	拷贝: 0	分配: 272730
Plain	565ms	拷贝: 0	分配: 272730
COW_Unsafe	683ms	拷贝: 0	分配: 272730
COW_AtomicInt	804ms	拷贝: 0	分配: 272730
COW_AtomicInt2	844ms	拷贝: 0	分配: 363640
COW_CritSec	825ms	拷贝: 0	分配: 272730
COW_Mutex	8419ms	拷贝: 0	分配: 272730

执行 1,000,000 次重复操作，字符串长度为 50:

Plain_FastAlloc	218ms	拷贝: 0	分配: 137262
Plain	354ms	拷贝: 0	分配: 137262
COW_Unsafe	474ms	拷贝: 0	分配: 137262
COW_AtomicInt	588ms	拷贝: 0	分配: 137262
COW_AtomicInt2	536ms	拷贝: 0	分配: 156871
COW_CritSec	607ms	拷贝: 0	分配: 137262
COW_Mutex	7614ms	拷贝: 0	分配: 137262

执行 1,000,000 次重复操作，字符串长度为 100:

Plain_FastAlloc	182ms	拷贝: 0	分配: 79216
Plain	257ms	拷贝: 0	分配: 79216
COW_Unsafe	382ms	拷贝: 0	分配: 79216
COW_AtomicInt	492ms	拷贝: 0	分配: 79216
COW_AtomicInt2	420ms	拷贝: 0	分配: 89118
COW_CritSec	535ms	拷贝: 0	分配: 79216
COW_Mutex	7810ms	拷贝: 0	分配: 79216

执行 1,000,000 次重复操作，字符串长度为 250:

Plain_FastAlloc	152ms	拷贝: 0	分配: 43839
Plain	210ms	拷贝: 0	分配: 43839
COW_Unsafe	331ms	拷贝: 0	分配: 43839
COW_AtomicInt	438ms	拷贝: 0	分配: 43839
COW_AtomicInt2	366ms	拷贝: 0	分配: 47825
COW_CritSec	485ms	拷贝: 0	分配: 43839
COW_Mutex	7358ms	拷贝: 0	分配: 43839

执行 1,000,000 次重复操作，字符串长度为 1,000:

Plain_FastAlloc	123ms	拷贝: 0	分配: 14000
Plain	149ms	拷贝: 0	分配: 14000
COW_Unsafe	275ms	拷贝: 0	分配: 14000
COW_AtomicInt	384ms	拷贝: 0	分配: 14000
COW_AtomicInt2	299ms	拷贝: 0	分配: 15000
COW_CritSec	421ms	拷贝: 0	分配: 14000
COW_Mutex	7265ms	拷贝: 0	分配: 14000

执行 1,000,000 次重复操作，字符串长度为 2,500:

Plain_FastAlloc	122ms	拷贝: 0	分配: 6416
Plain	148ms	拷贝: 0	分配: 6416
COW_Unsafe	279ms	拷贝: 0	分配: 6416
COW_AtomicInt	380ms	拷贝: 0	分配: 6416
COW_AtomicInt2	304ms	拷贝: 0	分配: 6817
COW_CritSec	405ms	拷贝: 0	分配: 6416
COW_Mutex	7281ms	拷贝: 0	分配: 6416

测试 operator[](): 一个可能会修改、但永远不会真正修改的操作

说明：

- Plain 的表现总是大大优于 COW.
- 测试结果和字符串长度无关。
- COW_AtomicInt 和 Plain 在开销上的差值近似于常数，每 1,000,000 个操作相差约 280ms。
- COW_AtomicInt2 在这一测试场合下表现较好，但在总体上，COW_AtomicInt 表现更佳。因此我将注意力集中在和 Plain 的对比上。

[重复次数是前面测试的 10 倍] 执行 10,000,000 次重复操作，字符串长度为 10:

Plain_FastAlloc	3ms	拷贝: 0	分配: 3 [*]
Plain	2ms	拷贝: 0	分配: 3 [*]
COW_Unsafe	1698ms	拷贝: 0	分配: 3
COW_AtomicInt	2833ms	拷贝: 0	分配: 3
COW_AtomicInt2	2112ms	拷贝: 0	分配: 4
COW_CritSec	3587ms	拷贝: 0	分配: 3
COW_Mutex	71787ms	拷贝: 0	分配: 3

[*] 测试结果在错误范围之内。二者都在 0ms 至 9ms 这一范围之间。

测试各种整数递增/递减操作

测试简述：

- “plain” 对普通的非 volatile int 执行操作。
- “volatile” 是使用 volatile int 的唯一例子。
- “atomic” 使用 Win32 InterlockedXxx() 操作。
- “atomic_asm” 使用内嵌 x86 汇编语言中的锁定整数操作 (locked integer operation)。

说明：

- 较之 ++volatile 或未优化的 ++plain, ++atomic 的耗时只是它们的三倍。
- ++atomic 不会招致函数调用的开销。

[重复次数是前面测试的 100 倍] 重复执行 100,000,000 次整数操作：

```

++plain    2404ms, counter=100000000
--plain    2399ms, counter=0

++volatile   2400ms, counter=100000000
--volatile   2405ms, counter=0

++atomic    7480ms, counter=100000000
--atomic    9340ms, counter=0

```

```
++atomic_asm    8881ms, counter=100000000
--atomic_asm    10964ms, counter=0
```

这里还有一点补充说明。针对 `IntAtomicIncrement()` 的各种不同的 x86 汇编实现，我测试了其耗时对比。时间的测试基于和上面相同的条件，我们可以直接对其进行比较（上面的测试实际上使用的就是这个实现）：

指令	耗时
<code>_asm mov eax, 1</code>	
<code>_asm lock xadd i, eax</code>	
<code>_asm mov result, eax</code>	$\sim 11000\text{ms}$
<code>_asm mov eax, 1</code>	
<code>_asm lock xadd i, eax</code>	$\sim 10400\text{ms}$
<code>_asm lock inc i</code>	$\sim 8900\text{ms}$
<code>_asm inc i</code>	$\sim 2400\text{ms}$

注意，非原子操作（non-atomic）版本的表现要好得多，其性能对应于“plain”。

结论：x86 的 LOCK 指令的确会带来开销，即使在单 CPU 机器上也是如此。这很自然，而且也在意料之中。但我还是要特别指出这一点，因为过去有人声称，使用 LOCK 指令不会造成性能上的差异。

最后请注意，可以很明显地看到：Win32 原子整数函数不会招致函数调用的开销，所以，绝对不要猜测——请实际测试！

测试程序

我编写的测试程序和 `String` 的实现代码在线提供于 *More Exceptional C++* 的主页上，即 <http://www.gotw.ca/publications/mxc++.htm>。

参考文献

- [Alexandrescu00a] A.Alexandrescu. “Traits on Steroids”(*C++ Report* 12(6), June 2000).
- [Alexandrescu00b] A.Alexandrescu. “Mappings Between Types and Values” (*C/C++ Users Journal* C++ Experts Forum, 18(10), October 2000). Available online at http://www.gotw.ca/publications/mxc++/aa_mappings.htm.
- [Alexandrescu01] A.Alexandrescu. *Modern C++ Design*(Addison-Wesley,2001).
- [C++98] ISO/IEC 14882:1998(E), *Programming Language-C++*(ISO and ANSI C++ standard).
- [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*(<http://www.gotw.ca/gotw>).
- [Henney00] K.Henney. “From Mechanism to Method: Substitutability”(*C++ Report*, 12(5), May 2000). Available online at http://www.gotw.ca/publications/mxc++/kh_substitutability.htm.
- [Koenig99] A.Koenig. “Changing Containers Iteratively”(*C++ Report*, 11(2), February 1999).
- [Knuth97] D.Knuth. *The Art of Computer Programming, Vol.1, Fundamental Algorithms*, 3^d Edition(Addison-Wesley, 1997).
- [Lippman98] S.Lippman and J.Lajoie. *C++ Primer*, 3^d Edition(Addison-Wesley, 1998).
- [Liskov88] B.Liskov. “Data Abstraction and Hierarchy”(*SIGPLAN Notices*, 23(5), May 1988).
- [Meyers96] S.Meyers. *More Effective C++* (Addison-Wesley, 1996).
- [Meyers97] S.Meyers. *Effective C++ 2^d Edition*(Addison-Wesley, 1997).
- [Meyers99] S.Meyers. *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*(Addison-Wesley, 1999).
- [Meyers01] S.Meyers. *Effective STL*(Addison-Wesley, 2001).
- [Murray93] R.Murray. *C++ Strategies and Tactics*(Addison-Wesley, 1993), Pages 70-72.
- [Niec00] T.Niec. “Optimizing Substring Operations in String Classes”(*C/C++ Users Journal*, 18(10), October 2000).

- [Stroustrup94] B.Stroustrup. *The Design and Evolution of C++*, Section 15.4.2(Addison-Wesley, 1994).
- [Stroustrup00] B.Stroustrup. *The C++ Programming Language, Special Edition* (Addison-Wesley, 2000).
- [StroustrupFAQ] B.Stroustrup. “C++ Style and Technique FAQ.” Available online at http://www.gotw.ca/publications/mxc++/bs_constraints.htm.
- [Sutter00] H.Sutter. *Exceptional C++*(Addison-Wesley, 2000).
- [Sutter01] H.Sutter. “Virtuality”(*C/C++ Users Journal*, 19(9), September 2001).

索引

#define, see macros
#pragma, 218

A

AboutToModify, 91-114
Abrahams, David, 138, 140, 142
accumulate, 77-78
ACE, 104
Adams, Douglas, reference to, 22
Adapter pattern, 179-180
advance, example use, 10
Alexandrescu, Andrei, 25, 271
Append, 86-103, 263-269
array(s),
 auto_ptr and, see auto_ptr, arrays and
 vector and, see vector, C arrays and
 zero-length, 177-178
Arrays, 65-68
assignment, templated, see template(s)
 assignment
associative containers, see containers, associative
atomic integer operations, 107-114, 256-258, 269
auto_array, 178-179
auto_ptr, 135-148, 175-199
 arrays and, 175-182
 example use, 93
 members, issues regarding, 148, 182-187
 Pimpl Idiom and, see Pimpl Idiom, auto_ptr
 and
 swap and, 145
 vector versus, 181-182
auto_ptr_new, 137-140

B

basic_ios, 3-4
 rdbuf, 3-4
basic_istream, 1-5
basic_ostream, 1-5, 20
basic_string, 20, 113, 195, 197
 copy-on-write and, 259-260
 see also copy-on-write
 reputation for slowness undeserved, 113, 248
bazooka, const_cast, 58
Bester, Alfred, 55
bitset, 41, 43
bitvector, 43
bool,
 vector and, see vector, vector<bool>
bpred, 11-12
buffergrowthstrategy, see String.growthstrategy

C

C array
 vector and, see vector, C array and
C/C++ Users Journal, x
C++ standard, the, 271
C++ Report, x
capacity, see vector, capacity
Cargill Widget Example, 141-149
char_traits, 5, 20, 195
<chandler> style, 237-238
ChoosePeg, 71-73, 76
ci_char_traits, alluded to, 5
cin, 2
Clamage, Steve, xii, 140

class,
 local, 207-209
 nested, 207-209
 Clone, 19-32, 193-199
 Clonable, 19, 25-29
 ColorMatch, 77-79
 Colossal Cave adventure, reference to, 22
 Combination, 76
 concurrency, see multithreading
 conservation of energy, 93
 const_cast, 53, 55, 57-58
 Constraints, 24-28
 constructor,
 exceptions, see exception(s), constructor
 templated, see template(s), construction
 use to contain requirements-enforcing code, 22-23
 container requirements, 36-45
 container(s),
 associative, see set; map
 proxied, 41-45
 taking pointers and references into, 38
 vector<bool>, see vector, vector<bool>
 containment, see delegation
 conversion operators, see operator(s), conversion
 copy, example use, 9, 87-88, 92-93
 copy initialization, see initialization,
 direct versus copy
 copy-on-write,
 see also String, Optimized::String
 basic_string and, 259-261
 historically popular, 261
 multithreading and, 104-114, 248, 253-269
 mutating or possibly-mutating operations and, 251
 optimizing allocation versus char copying, 259
 performance, 111-114
 sample timings of, 263-269
 count, 79
 CountColor, 72, 74
 CountedPtr, 17-18
 CountPlace, 71, 73
 coupling, 151-153
 cout, 2
 COW, see copy-on-write
 crashes, see intermittent crashes
 CreateFrom, 194-199
 Customer, 58

D
 Death, Dreaded Diamond of, 156

debugger, launching, see DieDieDie
 declarations,
 forward, 226-228
 recursive, 201-206
 Decorator pattern, 170
 Decrement, 18
 deep copy, 108-109
 see also copy-on-write
 Del Rey, Lester, 55
 delete
 array from and auto_ptr, 176-182
 dependent names, 33
 deque, 1, 46-53
 deque<bool>, 45
 derivation, see inheritance
 destructor(s)
 use to contain requirements-enforcing code, 22-23
 pre-destructor, 131
 public versus virtual, 168-169
 Dewhurst, Steve, xii
 direct initialization, see initialization, direct
 versus copy
 DieDieDie, 171-172
 Dining Philosophers Problem, allusion to, 252
 distance, example use, 10
 double delete, examples of, 184
 Dr. Dobb's Journal, x
 Dreaded Diamond of Death, 156
 dynamic_cast, example use of, 160-162

E
 ECHO, 1-2
 effects, see also side effects
 encapsulation, 5-6, 20
 EnsureUnique, 256
 equal_to, example use, 77
 erase, 8, 46
 iterators and, see iterator(s), validity
 remove and, 8
 see also vector, erase
 Error, variant data representation using
 preprocessor, 217
 expressions, order of evaluation of, see order
 of evaluation
 exception(s),
 constructor, 115-126
 destructors and, 128, 130
 safety guarantees, see exception safety,
 guarantees

- thrown from base or member constructors, 116-126
exception safety, assignment and, 142, 146 coupling and, 151-153 design and, 147-148 guarantees, 142 Swap and, 142 extensibility, 5-6
- F**
 find, 43
 find_if, 12-17, 76
 FindCustomer, 47
 FlagNth, 7, 9-10, 15-17
 FlagNthImpl, 16-17
 flip, 42
 food, see Dining Philosophers Problem
 for_each, example use of, 72
 forward declarations, see declarations, forward
 forward iterators, see iterator(s), forward
 FPDoubleInt, 202
 French, gratuitous use of, x, 3, 77, 245
 friend,
 virtual function accessibility control using, 172, 174
 front, 42
 FuncPtr, 201-206
 function try blocks, see try, function try blocks
 functions, nested, see nested functions, simulating
- G**
 Gamma, Erich, 271
 generate, example use of, 71, 76
 GetError, 249, 251
 GotW, see Guru of the Week
 GreaterThan, 13
 GreaterThanFive, 12
 growth strategy for buffers, see String, growth strategy
 guarantees, exception safety, see exception safety, guarantees
 Guru of the Week, x, 245, 271
- H**
 Haldeman, Joe, 55
 Has-A, 149-153
 HasClone, 24
 header files
 <cheader>style, 237-238
 Heinlein, Robert A., 55, 114
 Helm, Richard, 271
 Henney, Kevlin, xii, 150, 271
 Hickin, John, 14
 "holey array", 125
 Holmes, Sherlock, reference to, 124
 Hyslop, Jim, xii
- I**
 ifstream, example use, 3-4
 IIITO, see Is-Implemented-In-Terms-Of
 increment(ing), see operator(s), ++
 Increment, 18
 indigestion, compiler, 136
 inheritance,
 delegation versus, 149-153
 multiple, 155-158
 simulating, 159-166
 Siamese Twin Problem, 162-166
 requiring, 19-32
 testing for, see also IsDerivedFrom
 unnecessary use of, 149-153
 virtual, 156
 initialization,
 direct versus copy, 224-226
 initializer lists,
 dangers of resource acquisition and, 123
 inline, 83-86
 inner_product, 77
 instantiated_type, 32-36
 instantiation of templates, see template(s), instantiation
 interface classes, 158
 intermittent crashes, 250
 IntAtomic..., see atomic integer operations
 integers,
 manipulating atomically, see atomic integer operations
 Is-A, 149-153
 see also Liskov Substitutability Principle(LSP)
 IsAnyElementGreaterThanFive, 12-13
 IsDerivedFrom, 25-29
 Is-Implemented-In-Terms-Of, 149-153
 IsOK, 118, 121
 Is-Substitutable-For-A, 149-153
 see also Liskov Substitutability Principle(LSP)
 istream, see basic_istream
 istream_with_assign, 2

i
iterator_traits, 20
iterator(s),
 distance and, 10
 forward, example use, 6-10
 modifying a set or map through an, 53-59
 pointers versus, 39
 random-access, 10
 ranges, 12
 traits, 10
 validity, 62-64

J
Johnson, Ralph, 271

K
Kanze, James, 140
Kdlin, John, 164
Koenig, Andrew, vii-viii, xii, 271
Knuth, Donald, 271

L
Lajoie, Josée, x, 271
Latin, gratuitous use of, 114, 212
Length, 95-114
less, 54
lifetime of objects, see **object(s), lifetime**
Lippman, Stan, x, 271
Liskov, Barbara, 149, 271
Liskov Substitutability Principle(LSP), 149-153
local class, see **class, local**
Lock, 251-252, 257
locking, see **copy-on-write**
LSP, see **Liskov Substitutability Principle(LSP)**

M
macros, 60, 215-222
make_pair, 53, 56
Mancl, Dennis, xii
map, 1, 53-59
 example use, 39
 find, 43
 keys and, 54-59
Mastermind, 69-81
max, 218-222
member template, see **template(s), member template**
 instantiation, see **template(s), instantiation**
Meyers, Scott, x, xii, 148, 271
MI, see **inheritance, multiple**
min, 79

modal operation, weaknesses of, 130
multimap, see **map**
multiple inheritance, see **inheritance, multiple**
multiset, see **set**
multithreading, see also **String, Optimized::String**
 optimization and, 103-114, 247-269
Murray, Rob, 271
mutable, 56-57
mutex, 107
Mutex, 107-113, 251
myproject_last.h, 243-244

N
namespace(s), 231-244
 file versus function scope, xi
 migrating to, 235-244
 qualification, xi
 use of xi
nested class, see **class, nested**
nested function, simulating, 206-215
Niec, Todd, 271
Niven, Larry, 55, 114
Nordberg, Henrik, 179
null pointer, 171-172
numeric_limits, 20

O
obfuscation, 3, 215
object(s),
 lifetime, 116-119, 138
 slicing, 192
 temporary, 138
ODR, see **One Definition Rule**
ofstream, example use, 3
omniORB, 104
omnithread, 104
One Definition Rule, 237
operator(s),
 *, 18, 40, 189-199
 &, 40
 <, 54
 ->, 18, 189-199
 ++, 59-64
 preincrement versus postincrement, 61
 (), 11-18, 32-36, 60, 72-74, 76
 [], 38, 42, 94-114
 conversion, 78
 optimization, 83-86

see also multithreading, optimization and inline and, see inline premature, 43-45

O

Optimized::String, see String, Optimized::String order of evaluation, 132-140

Original::String, see String, Original::String ostream, see basic_ostream overload resolution, 67-69 see also template(s), overloading nontemplate functions preferred, 68

P

pair, 39, 54 parentheses, macros and, 219 Parrot, 116-119

Pimpl Idiom, 120-121, 144-146, 189 auto_ptr and, 185 ValuePtr and, 187-199

Pink Floyd, 119

plus, example use, 77

pointer(s), into a string, see operator(s), [] iterators versus, 39 null, see null pointer unmanaged, 132-140 wild, see wild pointer polymorphism, using templates, 4-5 using virtual functions, 4-5

pop-front, 49

postincrement, see operator(s), ++ pred, 11-12

pre-destructors, 131 predicates, 11-18 binary, 11-14 copying, 16-17 order applied to elements in a range unspecified, 16-17 reference counting and, 16-18 state and, 14-18 unary, 11-14

preincrement, see operator(s), ++

P

Process, 4-5

Prompt, 76

protocol classes, 158

Proxied containers, see container(s), proxied puns, shameless, xii

pure virtual function, see virtual function, pure push_front, 49

Q

qsort, 202

R

RAII idiom, see resource acquisition is initialization idiom rand, example use of, 71, 76 rdbuf, see basic_ios, rdbuf ReadBuf, 162-166 recursion, declarations and, see declarations, recursive macros and, 221

reference, 42

reference counting, copy-on-write (COW) and, see copy-on-write predicates and, see predicates, reference counting and strings and, see copy-on-write reference(s), into a string, see operator(s), []

release, 189

remove, 6-10 erase and, 8 doesn't really, 1 remove-erase trick, 8 remove_if, 7-10, 15-17 remove_nth, 6, 9-10 reserve, see vector, reserve Reserve, 86-103, 108-109 reset, 189

resource acquisition is initialization idiom, 123, 139-140 initializer lists and, 123

return, illegal from constructor function try block handler, 120

reuse, 199

Rolling Stones, The, 119

Rose, 36

S

Schmidt, Bobby, 121

scissors, see also iterator(s), validity running with, iterator validity and, 62-64

sequence points, see order of evaluation set, 1, 53-59

keys and, 54-59
 set_terminate, 125
 set_unexpected, 125
 SetError, 249-253
 shallow copy, 93
 see also copy-on-write
 side effects, 61-64
 Siek, Jeremy, 24
 sizeof,
 array size and, 48
 effective abuse of, 25-27
 small-string optimization, 261
 smell, 36
 Siamese Twin Problem, see inheritance, multiple,
 Siamese Twin Problem
 slicing, see object(s), slicing
 Smith, E.E., 55
 Socrates, ix
 sort, example use of, 65-66
 SQL anecdote, 157
 srand, example use of, 71, 76
 standard C++, see C++ standard, the
 Stanislaus, 37
 state,
 predicates and, see predicates, state and
 state machines, 202-204
 State Pattern, 169
 stateful predicates, see predicates, state and
 stateless predicates, see predicates, state and
 std::, see namespace(s), qualification
 Stepanov, Alex, 24
 string, see basic_string
 String,
 copy-on-write, see String, Optimized::String
 growth strategy, 88-90
 Original::String, 86-90, 248
 Optimized::String, 90-114, 248-254
 unshareability, 97-103
 StringBuf, 91-103, 256-257
 strings,
 small-string optimization, 261
 Stroustrup, Bjarne, x, xii, 24, 271
 swap, 51
 auto_ptr and, 145
 Swap, 142, 189-199

T

TC, see Technical Corrigendum
 Technical Corrigendum, 48

template instantiation, see template(s), instantiation
 Template Method pattern, 171
 template(s),
 assignment, 191-193
 construction, 191-193
 instantiation, 21
 member template, example use, 9, 17
 overloading, 67-69
 default parameters and, 67-68
 polymorphism and, see polymorphism, using
 templates
 selecting alternative implementations, 29-32
 specialization, 64-69
 explicit, 65
 partial, 66
 traits templates, see traits
 temporary objects, see object(s), temporay
 terminate, 125, 127
 terse code, 2-3
 theft, grand, 186
 threads, see multithreading
 throw
 from constructor function try block handler, 120
 traffic, see also iterator(s), validity
 playing in,
 iterator validity and, 62-64
 traits, 19-32
 transform, example use of, 71-73
 trigraph, 222
 trilogy, must comprise four things, 168
 try
 function try blocks, 116-119, 123
 terminate and, 125
 typedef, 228-231
 recursion and, 204-205
 typename and, 35-36
 typename, xi-xii, 32-36
 typedef and, 35-36

U

Unadvise, 164
 unary_function, 12-14
 uncaught_exception, 126-131
 unexpected, 125
 universal character name, 222
 unshareable strings, see String, unshareability
 URL references, see Web references
 using-declarations, 231-233
 don't use in header files, 236-237

- using-directives, 233-234
don't use in header files, 236
- V**
- ValidateRequirements, 23-24, 27
value_type, 37-40
ValuePtr, 187-199
 deep copy using copy constructor, 190-193
 deep copy using traits, 193-199
 strict ownership, 188-189
- Van Vogt, A.E., 55
- Van Winkel, Jan Christiaan, xii
- vector, 1, 36-53
 bool and, see vector, vector<bool>
 C array and, 46-47, 181-182
 capacity, 49
 clearing, 52-53
 contiguity, 47-48
 erase, 51
 growth strategy, 88-90
 reserve, 46, 48-51
 shrink-to-fit, 50
 vector<bool>, 40-45
 alternatives, 44-45
- virtual functions,
 accessibility control using friend, 172-174
 polymorphism and, see polymorphism, using
 virtual functions
 pure, 167-172
 techniques, 167-174
- Vlissides, John, 271
- volatile, 113
- VPTraits, 195-199
- W**
- Web references, xii
- wild pointer, 314
- Woods and Crowther Colossal Cave adventure,
 reference to, 22
- X**
- X_base, 32-36
- XTraits, 13-32