第三版序

我在 1991 年写下了《Effective C++》的原始版本。时间走到了 1997 年，这本书迎来了她的第二个版本，我对书中的内容做出了重大的更新，但是，由于我并不想让熟悉第一版的读者感到困惑，因此我尽我最大努力保持了本书的现有结构：对于条目的标题，在所有的 50 个条目中，有 48 个与原始版本保持基本一致。打这样一个比方：如果这本书是一栋房子，那么第二版就好像是通过更换屋内的地毯、灯具，以及重新涂漆来使这栋老房子焕发新颜。

我写这本书的目标，是在尽量小的、可读的篇幅内，讲解清 C++最重要的编程守则。自 1991 年以来，C++的世界经历了巨大的变革，我在 15 年前确立的那些条目已经无法达成这一最初目标了。因此，在第三版中，我决定把这所老房子彻底推倒重建（我一直渴望能够摒弃一切从零开始）。在 1991 年时我们可以很自然的假设 C++程序员拥有 C 语言背景。但是现在，迁移到 C++的程序员很可能来自 Java 或者 C# 阵营。在 1991 年，继承和面向对象编程对于大多数程序员而言还是全新的概念。而现如今，这两个概念已经得到了广泛的应用。此时，异常处理、模板编程、和泛型编程等全新的领域才需要得到更多的指引。在 1991 年，“设计模式”还是一个全新的概念。而今，脱离设计模式来讨论软件系统几乎是不可能的。在 1991 年，人们针对 C++首个正式标准的创立工作才刚刚 4 开始。而现在，距首个标准创立已经八年之久，当前，标准的下一个版本的制定工作已经提上了日程。

为了解决这些变革所带来的影响，我尽量删除了先前版本中所有陈旧的内容，并且始终向我自己提问：“在 2005 年，学习 C++编程时最重要的建议是什么？”于是便形成了本书这一新版本中全新的条目体系。新版本中有两个全新设立的章节，分别介绍了资源管理和模板编程的内容。事实上，与模板相关的内容贯穿全书，因为这些内容几乎影响到了 C++的方方面面。本书中还包含一些新内容，包括处理异常的编程，对设计模式的应用，以及对全新的 TR1 库中新功能的使用。（条目 54 中将讨论 TR1。）众所周知，在单线程系统中能够完美使用的技术和手段，在多线程系统中可能不会奏效，因此，本书中超出半数的内容都是全新的。然而，第二版中大多数基本的信息仍然是很重要的，因此，我设法在新版中以各种形式保留了这些信息。（在附录 B 中你可以找到一个第二版与第三版条目的对照表。）

我已经尽我最大的努力让这本书趋于完美，但是我并不幻想它能够尽善尽美。如果你觉得本书中某些条目不适合作为普遍的编程建议；或者，如果你认为本书中所探究的某项任务可以通过一个更好的方法来达成，亦或，某一项或几项技术讨论不明确、不完整或者存在歧义，请你告诉我。如果你发现了本书中的错误，不论是何种类型的——技术上的、语法上的、排版上的，等等——请你也要告诉我。对于每一个有意义的错误，我会把第一位发现者的姓名添加进下一印刷批次中的“鸣谢”章节中，我将为此感到高兴。

即使条目总数扩展到了 55，书中讲解的编程守则还远远没有做到面面俱到。优秀的编程守则可以在任意时刻对任意程序都奏效，然而规划出优秀的守则往往比看上去要困难得多。如果你有为本书添加更优秀的编程守则的建议，我将十分乐意倾听。

我维护了一个表格，表格中记录了本书自第一次印刷以来所有的改动，包括 bug 修复、对模糊概念的澄清，以及技术更新等。你可以在“Effective C++勘误表”网页（http://aristeia.com/BookErrata/ec++3e-errata.html）中得到这一表格。如果你希望在我更新这一表格时收到通知，请你加入我的邮件列表。我将适时为对我的专业工作感兴趣的朋友发送公告。请访问 http://aristeia.com/MailingList/来获取更多细节。

斯考特 道格拉斯 梅耶斯　　俄勒冈州，斯坦佛大学

http://aristeia.com/　　2005 年四月

鸣谢

《Effective C++》已成书十五年，由于我是在撰写本书的五年前开始学习 C++的，因此可以说“Effective C++工程”已经运行了二十年了。期间，我收到了许多不同的见解、意见和建议，偶尔也会有一些异样的眼光。这些都帮助了《Effective C++》，使其不断进步，我对一切都心存感激。

至于我个人 C++的学习路线现已无法追溯，但是在我的记忆里有一个一般性信息的来源始终对我有很大的帮助，那就是 Usenet 的 C++新闻组，尤其是 `comp.lang.c++.moderated` 和 `comp.std.c++`。这些新闻组中的一些参与者非常善于对一些技术理念做出严谨而有深度的分析，本书中许多条目（也许是绝大多数）均受益于这些技术理念。

关于第三版中的新内容，我与 Steve Dewhurst 一同构思出了一系列可供选择的条目。在条目 11 中，通过拷贝并交换（copy-and-swap）来实现 `operator=`的思路来自 Herb Sutter 相关主题的著作，例如，《Exceptional C++》(Addison-Wesley，2000）的条目 13。RAII（参见本书条目 13）的概念来源于 Bjarne Stroustrup 的《The C++ Programming Language》（Addison-Wesley，2000）。条目 17 背后蕴含的思想来自 Boost 网站中 `shared_ptr` 网页中“最佳实践”（Best Practices）章节（http://www.boost.org/doc/libs/1_64_0/libs/smart_ptr/shared_ptr.htm#BestPractices），Herb Sutter 的《More Exceptional C++》（Addison-Wesley，2002）书中的条目 21 对这一思想进行了提炼。本书的条目 29 受到了 Herb Sutter 相关主题的诸多作品所带来的较为深远的影响，比如，《Exceptional C++》的条目 8 至 19，《More Exceptional C++》的条目 17 至 23，以及《Exceptional C++ Style》(Addison-Wesley，2005）的条目 11 至 13。David Abrahams 帮助我更好的理解了三大异常安全保证机制。本书条目 35 中的“NVI 惯用法”来自于 Herb Sutter 在《C/C++ User Journal》刊物中发表的关于“虚拟化”的专栏文章。另外，条目 35 中的“模板方法模式”和“策略模式”的概念来自《设计模式》（Addison-Wesley，1995）一书（该书作者：Erich Gamma，Richard Helm，Ralph Johnson，以及 John Vlissides）。本书条目 37 中关于运用 NVI 惯用法的思路来自于 Hendrik Schober。在条目 38 中，David Smallberg 促成了 `set` 模板的一个自定义实现版本。条目 39 中的推论“EBO 通常情况下不适用于多重继承”来源于 David Candevoorde 和 Nicolai M. Josuttis 的《C++

Templates》（Addison-Wesley，2003）一书。在条款 42 中，我对于 `typename` 最初的理解来自于 Greg Comeau 的“C++ and C FAQ”（http://www.comeaucomputing.com/techtalk/#typename），然后 Leor Zolman 帮助我认识到了我的理解是错误的。（这是我的错，与 Greg 无关。）条目 46 的中心思想源自 Dan Saks 的话：“广纳新朋。”条目 52 的末尾说明了这样一个理念：“如果你声明了 `operator new` 的一个版本，那么你就应该声明它所有的版本。”这一理念来自 Herb Sutter 的《Exceptional C++ Style》。David Abrahams 使我改进了我对 Boost 评审机制（在条目 55 中有总结）的理解。

上文中所提到的关于我从谁或者哪里学习到一些东西，并不意味着这个“谁”就是发明者，也不意味着这个“哪里”就是原始出处。

我的笔记告诉我，我同时也使用了来自 Steve Clamage，Antoine Trux，Timothy Knox，以及 Mike Kaelbling 的一些信息，但是遗憾的是，关于使用的方式以及来源，已无从查证。

Tom Cargill, Glenn Carroll, Tony Davis, Brian Kernighan, Jak Kirman, Doug Lea, Moises Lejter, Eugene Santos, Jr., John Shewchuk, John Stasko, Bjarne Stroustrup, Barbara Tilly, and Nancy L. Urbano 为本书第一版的手稿进行了审阅工作。我收到了来自以下朋友们对本书进行改进的建议：Nancy L. Urbano, Chris Treichel, David Corbin, Paul Gibson, Steve Vinoski, Tom Cargill, Neil Rhodes, David Bern, Russ Williams, Robert Brazile, Doug Morgan, Uwe Steinmüller, Mark Somer, Doug Moore, David Smallberg, Seth Meltzer, Oleg Shteynbuk, David Papurt, Tony Hansen, Peter McCluskey, Stefan Kuhlins, David Braunegg, Paul Chisholm, Adam Zell, Clovis Tondo, Mike Kaelbling, Natraj Kini, Lars Nyman, Greg Lutz, Tim Johnson, John Lakos, Roger Scott, Scott Frohman, Alan Rooks, Robert Poor, Eric Nagler, Antoine Trux, Cade Roux, Chandrika Gokul, Randy Mangoba，以及 Glenn Teitelbaum，在后续印刷批次皆予采纳。

Derek Bosch, Tim Johnson, Brian Kernighan, Junichi Kimura, Scott Lewandowski, Laura Michaels, David Smallberg, Clovis Tondo, Chris Van Wyk，以及 Oleg Zabluda 为本书第二版的手稿进行了审阅工作。后续的修订得益于 Daniel Steinberg, Arunprasad Marathe, Doug Stapp, Robert Hall, Cheryl Ferguson, Gary Bartlett,

Michael Tamm, Kendall Beaman, Eric Nagler, Max Hailperin, Joe Gottman, Richard Weeks, Valentin Bonnard, Jun He, Tim King, Don Maier, Ted Hill, Mark Harrison, Michael Rubenstein, Mark Rodgers, David Goh, Brenton Cooper, Andy Thomas-Cramer, Antoine Trux, John Wait, Brian Sharon, Liam Fitzpatrick, Bernd Mohr, Gary Yee, John O'Hanley, Brady Patterson, Christopher Peterson, Feliks Kluzniak, Isi Dunietz, Christopher Creutzi, Ian Cooper, Carl Harris, Mark Stickel, Clay Budin, Panayotis Matsinopoulos, David Smallberg, Herb Sutter, Pajo Misljencevic, Giulio Agostini, Fredrik Blomqvist, Jimmy Snyder, Byrial Jensen, Witold Kuzminski, Kazunobu Kuriyama, Michael Christensen, Jorge Yáñez Teruel, Mark Davis, Marty Rabinowitz, Ares Lagae，以及 Alexander Medvedev 的建议。

Brian Kernighan, Angelika Langer, Jesse Laeuchli, Roger E. Pedersen, Chris Van Wyk, Nicholas Stroustrup, and Hendrik Schober. Reviewers for a full draft were Leor Zolman, Mike Tsao, Eric Nagler, Gene Gutnik, David Abrahams, Gerhard Kreuzer, Drosos Kourounis, Brian Kernighan, Andrew Kirmse, Balog Pal, Emily Jagdhar, Eugene Kalenkovich, Mike Roze, Enrico Carrara, Benjamin Berck, Jack Reeves, Steve Schirripa, Martin Fallenstedt, Timothy Knox, Yun Bai, Michael Lanzetta, Philipp Janert, Guido Bartolucci, Michael Topic, Jeff Scherpelz, Chris Nauroth, Nishant Mittal, Jeff Somers, Hal Moroff, Vincent Manis, Brandon Chang, Greg Li, Jim Meehan, Alan Geller, Siddhartha Singh, Sam Lee, Sasan Dashtinezhad, Alex Marin, Steve Cai, Thomas Fruchterman, Cory Hicks, David Smallberg, Gunavardhan Kakulapati, Danny Rabbani, Jake Cohen, Hendrik Schober, Paco Viciana, Glenn Kennedy, Jeffrey D. Oldham, Nicholas Stroustrup, Matthew Wilson, Andrei Alexandrescu, Tim Johnson, Leon Matthews, Peter Dulimov，以及 Kevlin Henney 对本书第三版的一个原始的手稿进行了审阅。另外，Herb Sutter 和 Attila F. Fehér 对第三版中的一些单独条目进行了审阅。

审阅一份未经润饰的（很可能是不完整的）初稿是艰苦的，如果时间紧迫则尤甚。我依然要感谢有这么多人能为我承担起这份工作。

如果对于本书讨论的内容没有相关的知识背景，或者我们期望从手稿中找出**所有**的错误，审稿人的工作将变得难上加难。一些人竟依然敢于承担此项工作。Chrysta

Meadowbrooke 是本书的审稿人，她在审稿时事无巨细，使得本书中别人漏掉的诸多错误都能够浮出水面。

Leor Zolman 对于本书完整手稿中所有的代码示例都在不同的编译环境中进行了测试。并且在我修改之后又测试了一遍。如果书中的代码仍存在错误的话是我的责任，与 Leor 无关。

Karl Wiegers 和（特别是）Tim Johnson 为封底简介提供了快速的、有价值的反馈。

第一次印刷发行以来，我已经采纳了来自 Jason Ross, Robert Yokota, Bernhard Merkle, Attila Fehér, Gerhard Kreuzer, Marcin Sochacki, J. Daniel Smith, Idan Lupinsky, G. Wade Johnson, Clovis Tondo, Joshua Lehrer, T. David Hudson, Phillip Hellewell, Thomas Schell, Eldar Ronen, Ken Kobayashi, Cameron Mac Minn, John Hershberger, Alex Dumov, Vincent Stojanov, Andrew Henrick, Jiongxiong Chen, Balbir Singh, Fraser Ross, Niels Dekker, Harsh Gaurav Vangani, Vasily Poshehonov, Yukitoshi Fujimura, Alex Howlett, Ed Ji Xihuang. Mike Rizzi, Balog Pal, David Solomon, Tony Oliver, Martin Rottinger, Miaohua, Brian Johnson, Joe Suzow, Effeer Chen, Nate Kohl, Zachary Cohen, Owen Chu，以及 Molly Sharp 的修订建议。

John Wait 是本书前两版的编辑，但他“愚蠢地”把档期分配给另一个重量级项目。他的助理 Denise Mickelsen 用微笑婉拒了我数次的“骚扰”（至少我觉得她微笑了，尽管我没有真的见过她）。于是 Julie Nahil 前来救场成为此版本的责任编辑。她通过六周通宵达旦的工作紧张有序地完成了本书的编辑制作。John Fuller（她老板）和 Marty Rabinowitz（她老板的老板）也协助解决了一些制作上的问题。Vanessa Moore 的官方工作是解决 FrameMaker 问题和准备 PDF 版本，同时她还协助编写了附录 B, 并为其排版以适应封底内页印刷。Solveig Haugland 为目录进行了排版。Sandra Schroeder 和 Chuti Prasertsith 为本书设计了封面，每次我说“**这张那种**颜色的图片怎么样……”时好像 Chuti 都要修改封面。Chanda Leary-Coutu 解决了营销工作中的困难。

在我完成手稿的这几个月里，我经常在晚上观看电视剧《吸血鬼猎人巴菲》来给自己减压。只有极力克制我才不会在书中讲“巴菲语”。

Kathy Reed 曾在 1971 年向我讲授计算机编程，我很荣幸我们之间的友谊长存至今。Donald French 曾在 1989 年雇我和 Moises Lejter 创建 C++训练资料（也使得我**真正**了解了 C++），1991 年他邀请我参加了 Stratus Computer 的课程，期间学生们鼓励我进行写作，最终促成了本书的第一版本。Donald 还向我引荐了 John Wait，也就是当时本书的责任编辑。

我的妻子，Nancy L. Urbano，一直鼓励我写作，直至我完成了七本书、一部 CD 合集和一篇论文。她给予了我无限的耐心和宽容。没有她就不会有这些作品。

自始至终，我们家的狗狗 Persephone 一直是我无可替代的生活伴侣。但是令人伤心的是，本书成书的大部分时间她都在办公室里的一个骨灰盒里陪伴着我们。我们真的很想念她。

简介

学习一门编程语言的基础知识是一件事情，然而学习如何使用这一门语言来设计和实现**高效**的程序则是完全不同的另一件事。这一点在 C++领域尤为正确，C++因其生产力和表现力方面的卓越成绩而广受赞誉，如果得到正确使用，那么 C++可以使你的工作充满快感，大批各色各样的设计方案可以得到更直观的表达和更高效的实现，审慎选择并合理利用类、函数、和模板等技术可以使应用程序更加简洁、直观、高效，并且远离错误。一旦你掌握了相关手法，编写高效 C++程序并非难如登天。但是，如果无底线的滥用，你的 C++代码将会与可阅读性、可维护性、和可扩展性彻底无缘，并且终日与各种错误相伴。

本书的目的是向你展示如何**高效地**使用 C++。我将假设你事先知道 C++是一门**编程语言**，并且你拥有一定的 C++编程经验。本书中我将为你提供一个向导，来指引你如何使用这门编程语言让你的软件做到可阅读、可维护、可移植、可扩展、高效、并且能够按照预期运行。

我所提供的编程建议大致可以分为两类：一是基本的设计策略，二是具体语言特性的细节研究。关于设计我们讨论的中心是“在使用 C++来完成某些任务时，如何在不同的方案中做出正确选择”。使用继承还是模板？使用公共继承还是私有继承？使用私有继承还是组合？使用成员函数还是非成员函数？传值还是传引用？最初的选择至关重要，因为一个不良的方案所带来的后果很可能在开发过程展开很久后才会逐渐显现出来，此时，纠错的工作通常十分困难，且耗时费力。

即使你已经十分了解你当前的任务，让程序完全按部就班的运行也并非易事。赋值运算符返回什么类型才合适？析构函数在什么时候应该是虚拟的？`operator new` 在无法找到足够的内存时应该怎样运行？努力解决这些细节问题是十分必要的，这是因为一旦你的程序留有隐患，那么通常总会导致运行过程出现意外，甚至会出现令人困惑的行为。这本书将帮助你避免类似事情的发生。

本书并不是一本完整的 C++参考教材。而是一本包含了 55 条特定的编程建议（我称之为“条目”）可以改善你的程序和设计的书。每个条目大致上是独立的，但是大多数条目中都会包含对其他条目的引用。因此，你可以这样来阅读本书：从一个你感兴趣的条目开始，然后跟随条目中的引用，一步步深入学习。

本书也不是一本 C++的入门书籍。比如，在第二章中，我希望告诉你所有关于构造函数、析构函数和赋值运算符可行的实现方式，但是我会假设你事先了解这些函数的用途和声明方式，即使不了解，你也知道如何通过其他渠道来学习，许多 C++书籍中包含这一类信息。

我写本书的目的是要强调 C++编程中那些容易被人忽视的部分。如果你希望了解 C++的方方面面你还是需要转向其他书籍。本书会告诉你如何将这些方方面面有机结合起来，从而编出更高效的程序。其他书籍会告诉你程序是如何编译的，而本书则会告诉你如何去避免那些编译器也会忽略的问题。

与此同时，本书中讨论的 C++仅限于**标准** C++。本书中只会使用官方语言标准包含的特性。由于可移植性是本书所考虑的关键因素之一，因此如果你想在本书中找到一些非跨平台的“黑客手段”，那你一定会失望了。

你也不会在这本书里找到 C++的“不二法门”，让 C++程序处处完美的终极方案是不存在的。本书中的各个条目为你提供了各类编程向导，包括如何开发出更好的设计方案，如何避免一些常见的错误，如何使程序得到更高的性能，等等，但是这些向导中没有一条是普遍使用的。软件设计和实现是一项复杂的任务，这项任务与硬件、操作系统和应用程序层面的限制有着千丝万缕的联系，因此我所能做到最好的，就是为你提供编程向导，让你能够创造出更优秀的程序来。

如果你无时不刻都遵守这些编程守则，那么你就不大可能会落入 C++四周的一些较为普通的陷阱里，但是，守则是有例外的，这是由它的自然属性决定的。这也就是为什么每个条目中都有解释说明的文字。而且解释说明才是本书最重要的部分。只有你明白了每个条目背后蕴含的根本原理后，对于你正在开发的软件，以及你正面对的独一无二的难题，你才能做出正确的选择。

本书的最佳用途是深入了解 C++的行为方式，以及 C++表现出某些行为的内在原因，如何合理利用某些 C++的行为以发挥出它们的优势。盲目的滥用本书中某些条目很显然是不好的。但是与此同时，如果没有合理的理由，你最好还是不要破坏书中任意一项编程守则。

术语

以下是一个小型的 C++词汇表，每一位 C++程序员都应该了解它。下文中的一些术语十分重要，我们有必要在此对它们的意义达成一致。

【声明，declaration】一条声明告诉编译器某个东西的名字和类型。下面是一些声明：

```
extern int x;                           // 对象声明

std::size_t numDigits(int number);      // 函数声明

class Widget;                           // 类声明

template<typename T>                    // 模板声明
class GraphNode;                        // ("typename"的用法另请参见条目 42)
```

请留意上文中的 x，即使它是一个内建数据类型变量，我仍将其称为一个"对象"。有一些同仁认为只有用户自定义数据类型变量才能称为"对象"，但是我并不会这样做。同时请留意，numDigits 函数的返回值类型是 std::size_t，也就是说 size_t 类型在 std 名字空间中。几乎所有的 C++标准库的内容都位于 std 名字空间中。然而，由于 C 语言的标准库（更确切的说是 C89 的标准库）对于 C++仍然适用，因此从 C 语言继承而来的一些名字符号（比如 size_t）则可能出现三种情况，即：存在于全局，或存在于 std 中，或两者都有，这取决于你的程序包含了哪个头文件。本书中我假设所有程序包含的都是 C++的头文件，这也就是上文中我使用 std::size_t 而没有直接使用 size_t 的原因。当本书的行文中凡是提及标准库的某些组件时，我一般都会省略 std，这一点基于你已经了解诸如 size_t、vector 和 cout 等等就是定义在 std 中的。而在代码示例中，我将始终使用 std::，这是因为在真实代码的环境下，如果不这样做的话是无法通过编译的。

顺便说一句，size_t 仅仅是一个 typedef，它代替的是某些 C++用来计数的无符号类型数字（例如，在一个基于 char*的字符串内的字符数，一个 STL 容器内元素的数量，等等。）。与此同时，vector、deque 和 string 的 operator[]

函数的参数类型也是 size_t，在我们定义自己的 operator[]时也应该遵从这一惯例（条目 3）。

【签名，signature】每个函数在其声明语句中都可呈现出它自身的签名，也就是函数的参数表及返回值类型。我们说函数的签名与它的类型是同一个含义。比如，numDigits 的签名为：std::size_t (int)，也就是说，“该函数取一个 int 参数，返回一个 std::size_t 类型的值。”官方 C++对于“签名”的定义中并不包含函数的返回值，但是本书中，将返回值看做签名的一部分是更有实际意义的。

【定义，definition】定义向编译器提供声明语句中所省略的具体细节。对于对象而言，定义将为这个对象申请内存空间。对于函数和函数模板而言，定义为它们提供具体的代码。而对于类或者类模板而言，定义将列举出类或模板的所有成员：

```
int x;                                  // 对象定义

std::size_t numDigits(int number) // 函数定义。
{                                       // （此函数返回参数 number 的数字位数。）
  std::size_t digitsSoFar = 1;
  while ((number /= 10) != 0) ++digitsSoFar;
  return digitsSoFar;
}

class Widget {                          // 类定义
public:
  Widget();
  ~Widget();
  ...
};

template<typename T>                    // 模板定义
class GraphNode {
public:
  GraphNode();
  ~GraphNode();
  ...
};
```

【初始化，initialization】初始化是为一个对象赋初值的过程。对于用户定义类型的对象而言，初始化工作由构造函数完成。

【默认构造函数，default constructor】默认构造函数是可以不调用任何参数的构造函数。这类构造函数或者没有参数，或者所有参数都拥有一个默认值：

```
class A {
public:
  A();                                  // 默认构造函数
};

class B {
public:
  explicit B(int x = 0, bool b = true);// 默认构造函数;
};                                      // 关于"explicit"的信息请参见下文

class C {
public:
  explicit C(int x);                    // 不是默认构造函数
};
```

这里 B 类和 C 类的构造函数采用了显式（explicit）声明。这样做保证了这些类在能够做显式类型转换的同时，可以避免隐式类型转换：

```
void doSomething(B bObject);        // 一个函数:接收一个 B 类型的参数.

B bObj1;                            // 一个 B 类型的对象

doSomething(bObj1);                 // 正确，将一个 B 对象传递给 doSomeThing

B bObj2(28);                        // 正确，使用整形数字 28 来创建一个 B 对象
                                    // （参数 b 默认为 true）

doSomething(28);                    // 错误！ doSomething 接受 B 类型对象
                                    // 不接受 int 对象
                                    // 且不存在 int 到 B 的隐式转换
```

```
doSomething(B(28));                    // 正确,使用 B 的构造函数
                                       // 此次调用将一个 int 对象
                                       // 显式转换为 B 对象(也就是"转型")
                                       // (关于转型的信息参见条目 27)
```

通常我们更推荐使用显式声明的构造函数,这是因为这样做可以避免编译器进行意外的(通常是非计划中的)类型转换。除非有充分的理由让构造函数能够进行隐式类型转换,一般情况下我都会将构造函数声明为 `explicit` 的。我也希望你能够遵从相同的策略。

请注意,上文的实例中我对转型的参数做了加粗处理。整本书中,对于一些特别重要的内容,我都会用相同的加粗手法来引起你的注意力。(我也会加粗章节的数字,但仅仅是因为这样看上去很酷。)

【拷贝构造函数,copy constructor】拷贝构造函数用于使用同一类型的另一个对象来给当前对象赋值,

【拷贝赋值运算符,copy assignment operator】拷贝赋值运算符用于把同一类型的一个对象的值拷贝到另一个对象中。

```
class Widget {
public:
  Widget();                                // 默认构造函数
  Widget(const Widget& rhs);               // 拷贝构造函数
  Widget& operator=(const Widget& rhs);    // 拷贝赋值运算符
  ...
};

Widget w1;                                 // 调用默认构造函数

Widget w2(w1);                             // 调用拷贝构造函数

w1 = w2;                                   // 调用拷贝赋值运算符
```

上文中有一个语句看上去像是在进行赋值操作,当你看到它时请仔细阅读,这是因为"="的语法也可以用于调用拷贝构造函数:

```
Widget w3 = w2;                    // 调用拷贝构造函数！
```

幸运的是，我们很容易区分拷贝构造与拷贝赋值之间的不同。如果当前操作正在定义一个新的对象时（如上述代码示例中的 w3），此时我们应该调用它的构造函数，这个定义过程不应该是一次赋值。如果当前操作没有定义一个新的对象（如上述代码示例中的“w1 = w2”语句），则不应涉及构造函数的调用，则此处应该是一次赋值操作。

拷贝构造函数是一个尤为重要的函数，这是因为它定义了某类的对象是如何进行传值操作的。比如，请看下面的代码：

```
bool hasAcceptableQuality(Widget w);   // Widget 是否符合要求


...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

参数 w 通过传值方式传给 hasAcceptableQuality 函数，因此在上述的调用中，aWidget 被复制到 w。这一复制工作是由 Widget 的拷贝构造函数完成的。“传值”就意味着“调用拷贝构造函数”。（然而，通过传值方式传递用户定义类型的参数一般情况下并不是一个好方法。传递 const 引用通常是一个更好的选择。参见条目 20 获取更多信息。）

【STL】STL 即标准模板库（Standard Template Library），C++标准库中专门用来定义容器（比如 vector、list、set、map，等等）、迭代器（比如 vector<int>::iterator、set<string>::iterator，等等）、算法（比如 for_each、find、sort，等等），以及其他相关功能的部分。这些相关功能大多属于“函数对象”（像函数一样运行的对象）有关。函数对象来源于重载了函数调用操作符（即 operator()函数）的类。如果你对 STL 不甚了解，那么你在阅读本书的同时手边也要有一本正统的 C++参考书，这是因为 STL 实在是太棒了，我不得不在本书中将其充分利用起来。一旦你开始接触它，你很快就会和我一样体会到它的强大。

【未定义行为，undefined behavior】从 C#或 Java 等其他语言转向 C++的程序员们可能会对这个概念感到惊讶。基于很多原因，C++中某些场景下的行为确确实实

是不确定的：你无法准确的预测出运行时将会发生什么。以下是两段存在未定义行为的代码示例：

```
int *p = 0;                          // p 是一个空指针

std::cout << *p;                     // 空指针解引用
                                     // 产生未定义行为

char name[] = "Darla";               // name 是一个 char 数组
                                     // 长度为 6
                                     // （别忘了数组末尾的'\0'!）

char c = name[10];                   // 非法引用：数组角标越界
                                     // 产生未定义行为
```

为了强调未定义行为带来的后果有多么灾难性，编程大牛通常会告诉我们未定义行为有可能擦除硬盘上的信息。这不是耸人听闻：一个存在未定义行为的程序真的可能会擦除硬盘。但是这个可能性很小。更多情况下是你的程序可能运行的很不稳定：有时可以正常运行，有时则会崩溃，还有时会产生错误的结果。高效的 C++程序员会竭尽所能来避免未定义行为。本书中，一些存在未定义行为问题的地方我都做出了明确的说明，你应该特别留意。

【接口，interface】这是另一个会让从其他语言转向 C++的程序员困惑的概念。Java 和.NET 中，接口是语言的基本组成部分，但 C++在语言层面并没有接口的概念，我在条目 31 中讲解了如何在 C++中模拟出接口的功能。当我使用“接口”这一术语时，我一般讲的都是函数的签名，或是某个类中可访问的元素（比如类的“public 接口”，“protected 接口”，或“private 接口”），亦或是对某个模板的类参数必须有效的表达式。也就是说，我实际上是将接口做为一个一般的设计理念来讨论的。

【客户，client】一个“客户”指的是调用你编写的某段代码（通常是接口）的某位程序员或某段程序。举例说，一个函数的客户就是它的使用者，即调用该函数（或取得该函数的地址）的那一段代码，也包括编写或维护这些代码的程序员。当讨论客户的概念时，我一般会针对程序员，这是因为程序员是人，人都会感到困惑，会被误导，或者对糟糕的接口感到厌烦。但是他们写的程序则不会这样。

你可能还没有为客户着想的习惯，但是我会不惜花费大量时间来说服你尽可能使客户的工作更轻松。退一万步说，你也会成为其他人开发的软件系统的客户。你不也会希望其他人为你编写的系统能够更易用吗？与此同时，在某些场合你几乎一定会碰到“成为自己的客户”这种情况（也就是调用你自己编写的代码），在这种情况到来时，如果你在设计接口时充分考虑了客户的因素，那么你将会感到十分庆幸。

本书中，对于函数和函数模板的区别，以及类和类模板的区别，我通常会选择忽略它们。这是因为通常情况下，对于前者奏效的东西对后者同样奏效。在前后不一致的情况下，我将为上述两项区别做出详细区分。

命名的惯例

我尝试过为对象、类、函数、模板，等等，选择一些富有含义的名字。但是我选用的这些名字所包含的意义可能不会那么直观明显。比如，`lhs` 和 `rhs` 是我最喜欢用的参数名中的两个。它们分别表示“左侧（left-hand side）”和“右侧（right-hand side）”的意思。我通常在实现二元运算符（比如 `operator==`和 `operator*`）时使用这两个名字作为参数名。比如，如果 `a` 和 `b` 是表示有理数的对象，并且两个有理数对象可以通过 `operator*`这样一个非成员函数进行乘法运算（条目 24 讲的就很接近这种情景），表达式：

```
a * b
```

与下文的函数调用是等价的：

```
operator*(a,b)
```

在条目 24 中，我是这样声明 `opartor*` 的：

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

如你所见，左操作数 `a` 将在函数中命名为 `lhs`，右操作数 `b` 则命名为 `rhs`。

对于成员函数而言，由于左参数通过 `this` 指针来表示，因此一些情况下我会单独使用 `rhs` 作为参数名。你可能注意到了在上文中一些 `Widget` 的成员函数声明中存在过这样的做法。哦，这也让我想起了，我会经常在在示例中使用 `Widget` 作为类名。“`Widget`”仅仅是我为示例类起的一个简单的名字而已，并不意味着任何事情。它与 GUI 中的“控件”没有关系。

对于指针的命名我通常会遵循下面的规律：如果一个指针指向 T 类型的对象，那么这个指针就命名为 pt，即“指向 T 的指针（pointer to T）。”以下是示例：

```
Widget *pw;                          // pw: 指针指向 Widget

class Airplane;
Airplane *pa;                        // pa: 指针指向 Airplane
class GameCharacter;
GameCharacter *pgc;                  // pgc: 指针指向 GameCharacter
```

对于引用，我也遵循同样的规律：rw 可能是一个指向 Widget 的引用，ra 是一个指向 Airplane 的引用。

我偶尔会在讨论成员函数时将其命名为 mf。

关于多线程

C++在语言层面没有线程的概念，即：没有任何意义上“并发”的概念。在 C++的标准库里也没有。到目前我们讨论的 C++范畴不存在多线程的程序。

然而，多线程程序确实是存在的。虽然本书中我们讨论的焦点在标准的、可移植的 C++上，但是多数程序员在实际工作中依然会面对线程安全问题，多线程是一个不可忽略的因素。至于如何填平遵循 C++标准与尊重实际情况二者之间的鸿沟，我是这样做的：对于由 C++构建的程序，如果我检测出某些内容可能在多线程环境下会产生问题，那么我将随时标记出这些内容。这并不意味着这本书会成为一本讲解 C++多线程编程的书籍，还差得远呢。实际情况是这样的：本书讨论的 C++编程在绝大多数情况下限制在单线程环境里，我们承认多线程的存在，并且，对于需要考虑多线程编程的程序员而言，他们很可能期望评估我在本主题的建议是优是劣，我也会尽可能指出哪些需要他们特别注意的地方。

如果你对多线程并不熟悉，或者你的工作涉及不到多线程，那么你大可以忽略本书中线程相关的内容。然而，如果你正在编写一个多线程的应用程序或者库，那么请记住，比起在使用 C++时你需要解决的一些初级问题，这里的建议会更深一步。

TR1 和 Boost

你会发现 TR1 和 Boost 的内容始终贯穿本书。二者都有一个单独的条目来深入讲解（条目 54 讲解 TR1，条目 55 讲解 Boost），但是，可惜这两个条目在本书的最后。（把它们放在最后可以让本书结构更合理，这是真的，我也尝试过把它们放在书中其他地方，但是并不理想。）如果你喜欢，你大可以现在就翻到那两个条目去阅读，但是如果你更喜欢循序渐进，下文的指示性摘要可以帮助你免除困惑：

- TR1（“技术报告 1”，“Technical Report 1”）是对即将添加进 C++标准库的新功能的规范。新功能以新的类和新的函数模板形式存在，比如哈希表、引用计数智能指针、正则表达式，等等。所有 TR1 的组件都放置在 `tr1` 名字空间下，而 `tr1` 位于 `std` 名字空间内部。

- Boost 是一个组织，一个网站（http://boost.org），它为程序员提供了一套可移植的、业内公认的、开源的 C++库。大多数 TR1 的功能都是基于 Boost 已有的特征开发的，与此同时，直到编译器提供商在最新的发行版本中将 TR1 引入 C++类库之前，Boost 网站可能一直是开发人员寻找 TR1 实现的第一站。Boost 提供了比 TR1 更多的内容，了解一下无论如何都是值得的。

第一章. 适应 C++

无论你有怎样的编程背景，都需要用一些时间和精力来适应 C++。 C++是一门强大的编程语言，它拥有数不尽的实用功能，但是在你能够熟练地驾御 C++的力量，并且恰当而高效地利用这些功能之前，首先你要适应 C++做事情的方式。这本书自始至终都是在介绍如何顺利地完成这一适应过程，但是相比较大多数议题而言，还有一些更为基础的内容，本章向大家介绍的就是这些更为基础的东西。

条目1： 把 C++看作多种语言的联合体

起初，C++仅仅是增补了面向对象特性的 C 语言。甚至 C++原始的名称都叫做“C with Classes”，即“使用类的 C 语言”，二者的继承关系可见一斑。

随着 C++逐渐成熟，它开始采纳一些与“使用类的 C 语言”不同的理念、特征以及编程策略，使其显得更加大胆前卫，更富有冒险精神。在组织各个函数时异常处理机制需要不同的处理方式（参见条目 29），模板为设计模式带来了全新的思维方式（参见条目 41），与此同时，STL 使 C++拥有了前所未有的可扩展性。

今天的 C++是一门**多范型编程语言**，它包含面向过程、面向对象、函数式编程、泛型、元编程等等特征。C++的强大和灵活几乎是无可比拟的，但这也会造成一些困惑。所有“恰当用法”的准则似乎都存在例外。那么我们该如何正确理解这样一门语言呢？

最简单的办法就是把 C++看作一个由若干门语言组成的联合体，而不是一门单一的语言。在某个特定的子语言中，规则就会趋于简单、直接，而且容易记忆。当你切换到另一门子语言时，规则也就相应地改变了。为了理解 C++，你必须能够辨别 C++所有主要的子语言。幸运的是，主要的子语言只有四门：

- **C**。尽管变革是深刻的，C++仍然基于 C 语言。程序块、语句、预处理器、内建数据类型、数组、指针，等等，所有都来自于 C。在许多情况下，C++为某些问题提供的解决方案要比 C 更优秀（比如条目 2（预处理器的替代方法）和条目 13（使用对象管理资源）），但是当你发现你正在使用“C++中的 C”这一

部分编写程序时，高效编程原则就会反映出 C 语言更多的局限所在：没有模板、没有异常处理、没有重载，等等。

- **面向对象的 C++**。这一部分的 C++就是“使用类的 C 语言”的一切：类（包括构造函数和析构函数）、封装、继承、多态、虚函数（动态绑定），等等。这是 C++中面向对象的经典准则得到最为直接的应用的那一部分。

- **包含模板的 C++**。这是 C++中泛型编程的一部分，这也是大多数程序员涉足最浅的部分。模板的概念遍及 C++的方方面面，因此，某些优秀的编程守则中有一些特定的、仅针对模板的段落也不足为奇。（比如，条目 46 中介绍的在调用模板函数时如何简化类型转换）。事实上，模板如此之强大，它足以带来一个全新的编程范型：模板元编程(template meta-programming,简称 TMP)。条目 48 是对 TMP 的一个简介，然而除非你是一个狂热的“模板迷”，你大可不必投入过多精力。主流 C++编程很少涉及到 TMP 规则。

- **STL**。顾名思义，STL 是一个模板库，但是它是一个非常特别的模板库。它将容器、迭代器、算法、函数对象之间的约定十分优雅的相互协调在一起，当然模板和库也可以基于其它的理念来构建。STL 有自己独特的解决问题的方法，当你使用 STL 编程时，你必须要遵循它的约定。

时刻地对这四门子语言保持头脑清醒，当你从一门子语言切换到另一门时，高效的程序会要求你必须更改当前策略，遇到这种情况时请不要大惊小怪。比如说，对内建（比如类似 C 语言的）类型而言，传值要比传引用更高效，但是当你从“C++中的 C”迁移到“面向对象的 C++”后，构造函数和析构函数的存在就意味着传递 `const` 引用会更好。在使用“包含模板的 C++”时这一点尤其正确，因为你根本就不知道当前正在处理的对象是什么类型。然而当开始使用 STL 时，迭代器和函数对象都是基于 C 语言中的指针机制创建的，因此对于迭代器和函数对象而言，C 语言的传值规则又再次奏效了。（关于各种传参方法方案选择的细节，参见条目 20。）

综上，C++并不是一门仅仅拥有一套规则的单一化编程语言，它是四门子语言的联合体，每门子语言都有自己的约定。对这四门子语言时刻保持清醒，你会发现 C++并没有那么难于理解。

时刻牢记

- C++的高效编程守则不是一成不变的，它根据你正在使用的那一部分 C++而改变。

条目2： 多用 const、enum、inline，少用#define

这一条目似乎叫做“多用编译器，少用预处理器”更恰当，因为#define 的内容不应该属于语言**自身**的范畴。这是#define 的众多问题之一，请看下面的代码：

```
#define ASPECT_RATIO 1.653
```

编译器也许根本就接触不到 ASPECT_RATIO 这个符号名，它也许在编译器对源代码进行编译以前就被预处理器替换掉了。于是，ASPECT_RATIO 这一名字很可能不会被加入符号表。如果代码中使用了该常量，此时编译器产生的错误将会令人费解，因为出错信息只会涉及 1.653，而完全不会涉及 ASPECT_RATIO。如果 ASPECT_RATIO 是在某个其他人编写的头文件中定义的，那么对于它的来源你便一无所知，你将在跟踪这一数值上浪费很多时间。在符号调试器中同样的问题也会出现，原因和上述问题是一致的：你在编程时使用的名字可能不在符号表中。

解决的办法是：使用常量来代替宏定义：

```
const double AspectRatio = 1.653; // 宏的名字通常使用大写字母，
                                  // 于是常量使用这样的名字以示区别
```

作为语言层面的常量，AspectRatio 会确保被编译器所看到，并且肯定会进入符号表中。另外，对于浮点数的情况（比如上述示例），使用常量较#define 而言会生成更小的目标代码。这是由于预处理器会对目标代码中出现的所有宏 ASPECT_RATIO 复制出一份 1.653，然而使用常量 AspectRatio 时拷贝永远不会多于一份。

在使用常量代替#define 时有两个特殊情况值得注意。第一是定义指针常量的情形。因为常量一般都定义在头文件中（许多不同的代码文件会包含这些头文件），要将**指针本身**定义为 const 的，通常情况下也要将指针所指的内容定义为 const 的，这一点很重要。比如说，在头文件中定义一个基于 char*的字符串常量时，你需要写**两次** const：

```
const char * const authorName = "Scott Meyers";
```

对 const 的含义和用法的完整讨论，尤其是涉及指针的情形，请参见条目 3。但是，在这里有必要提醒你一下，使用 string 对象通常要比它的基于 char*的祖先好得多。因此，autherName 以这样的形式定义比较好：

```
const std::string authorName("Scott Meyers");
```

第二个特殊情况涉及类内部的常量。为了将常量的作用域限制在一个类里，你必须将这个常量作为类的成员；为了限制常量份数不超过一份，你必须将其声明为 static 成员：

```
class GamePlayer {
private:
  static const int NumTurns = 5;   // 常量声明
  int scores[NumTurns];             // 该常量的用法
  ...
};
```

上面你所看到的是 NumTurns 的**声明**，而不是定义。通常情况下，C++要求你为要用到的所有东西做出定义，但是这里有一个例外：类内部的静态常量如果是整型（比如整数、字符型、布尔型）则不需要定义。只要你不需要得到它们的地址，你可以声明它们、调用它们，而不提供定义。如果你需要得到类常量的地址；或者即使你不需要这一地址，而你的编译器错误地坚持你必须为这个常量做出定义，这两种情况下你应该以下面的形式提供其定义：

```
const int GamePlayer::NumTurns;    // 这是 NumTurns 的定义，
                                   // 下边会告诉你为什么不为其赋值。
```

你应该把这段代码放在一个实现文件中，而不是头文件中。这是因为类常量的初始值已经在其声明的时候给出了（比如说，NumTurns 在声明时就被初始化为 5），而在定义的时候不允许为其赋初值。

顺便要注意一下，使用#define 来创建一个类内部的静态常量是行不通的，这是因为#define 不考虑作用域的问题。一旦一个宏做出了定义，在编译时它将影响到所有其它代码（除非你在某处使用#undef 取消了这个宏的定义）。这不仅意味着#define 不能用来定义类内部的常量，同时还说明它无法为程序带来任何封装

效果，也就是说，“私有的#define”这类东西是不存在的。然而 const 数据成员可以得到封装，NumTurns 就是一个例子。

早期的编译器可能不会接受上面代码的语法，这是因为那时候在声明一个静态的类成员时为其赋初值是非法的。与此同时，只有整型数据和常量才可以在类内部进行初始化。在不能使用上述语法的情况下，你可以在定义的时候为其赋初值：

```
class CostEstimate {
private:
  static const double FudgeFactor; // 静态类常量的声明
  ...                              // 应在头文件中进行
};

const double                       //静态类常量的定义
  CostEstimate::FudgeFactor = 1.35; //应在实现文件中进行
```

以上几乎是你所要了解的全部内容了。只存在一种例外情形：在某个类的编译过程中，你可能需要这个类内部的一个常量的值，比如说上文中的 GamePlayer::scores 数组的声明（编译器可能会坚持在编译时了解数组的大小）。编译器在这时（错误地）禁止了为类内部的静态整型常量赋初值，那么有什么办法补救呢？你可以使用“enum 戏法手段”（这是爱称，不带有蔑视色彩）。这一技术利用了“在需要 int 值的地方，枚举类型数据的值也可以使用”这一事实，所以 GamePlayer 就可以这样定义了：

```
class GamePlayer {
private:
  enum { NumTurns = 5 };           // “enum 戏法”
                                   // 使 NumTurns 成为一个值为 5 的符号名

  int scores[NumTurns];            // 可以正常工作
  ...
};
```

从许多角度讲，了解 enum 戏法手段是很有好处的。首先，从某种程度讲，enum 戏法的行为更像#define 而不是 const，在某些情况下这更符合你的要求。比如说，你可以合法地取得一个 const 的地址，但是取 enum 的地址则是非法的，而

去取#define 的地址同样不合法。如果不想让其他人得到你的整型常量的指针或引用，那么使用枚举类型便是强制实施这一约束的一个很好的手段。（关于使用编码手段强制实现设计约束的更多信息，参见条目 18）与此同时，优秀的编译器不会为整型 const 对象分配内存，但是粗心大意的编译器也许会这么做，你也一定不会愿意为这类对象分配内存的。与#define 类似，enum 永远不会带来不必要的内存开销。

了解 enum 戏法的第二个用处纯粹是实用主义的。许多代码都在这样做，所以你看到它时必须要认得。事实上，enum 戏法是模板元编程的一个基本技术。（参见条目 48）

回到预处理器的问题，#defined 命令的另一个用法（这样做很不好，但这非常普遍）就是将宏定义得和函数一样，却没有函数调用的开销。下面例子中的宏定义使用 a 和 b 中更大的参数调用了一个名为 f 的函数：

```
// 使用 a 和 b 中更大的一个用于调用函数 f
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

这样的宏会带来数不清的缺点，想起来就让人头疼。

无论什么时候，只要你写下了这样的宏，你必须为宏内部所有的参数加上括号。否则，其他人在某些语句中调用这个宏的时候总会遇到麻烦。即使你做了正确的定义，古怪的事情也会发生：

```
int a = 5, b = 0;

CALL_WITH_MAX(++a, b);              // a 自加两次
CALL_WITH_MAX(++a, b+10);           // a 自加一次
```

在这里，调用 f 以前 a 自加的次数竟取决于它比较的对象！

幸运的是，你不需要把精力放在这些毫无意义的事情上。你可以使用内联函数的模板，此时程序仍拥有宏的高效，并且一切行为都是可预知的，类型安全的：

```
template<typename T>
inline void callWithMax(const T& a, const T& b)
// 由于我们不知道 T 的类型，因此要传递 const 引用。参见条目 20
```

```
{
  f(a > b ? a : b);
}
```

这一模板创建了一族函数，其中每一个函数都会得到同一类型的两个对象，并且使用其中较大的一个来调用 f。可以看到，在函数内部不需要为参数加括号，不需要担心参数会被操作的次数。与此同时，由于 callWithMax 是一个真实的函数，它遵循作用域和访问权的规则，比如我们可以顺理成章的让一个类拥有一个私有的内联函数。然而宏在这些问题上就望尘莫及了。

C++为你提供了 const、enum、inline 这些新特征，预处理器（尤其是#define）的作用就越来越小了，但是这并不是说可以完全抛弃它。#include 仍是程序中的主角，#ifdef/#ifndef 在控制编译过程还有着举足轻重的地位。说"预处理器该退休了"还为时过早，但是你还是要经常给它放放长假。

时刻牢记

- 对于简单的常量，应多用 const 对象或枚举类型数据，少用#define。
- 对于类似程序的宏，应多用内联函数，少用#define。

条目3： 尽可能使用 const

const 令人赞叹之处就是：你可以通过它来指定一个语义上的约束（一个特定的**不能够**更改的对象）这一约束由编译器来保证。通过一个 const，你可以告诉编译器和其他程序员，你的程序中有一个数值需要保持恒定不变。不管何时，当你需要这样一个数时，你都应确保对这一点做出声明，因为这样你便可以让编译器来协助你确保这一约束不被破坏。

const 关键字的用途十分广泛。在类的外部，你可以利用它定义全局的或者名字空间域的常量（参见条目 2），也可以通过添加 static 关键字来定义文件、函数、或者程序块域的对象。在类的内部，你可以使用它来定义静态的或者非静态的数据成员。对于指针，你可以指定一个指针是否是 const 的，其所指的数据是否是 const 的，或者两者都是 const，或者两者都不是。

```
char greeting[] = "Hello";
char *p = greeting;                 // 非 const 指针，非 const 数据
const char *p = greeting;           // 非 const 指针，const 数据
char * const p = greeting;          // const 指针，非 const 数据
const char * const p = greeting;    // const 指针，const 数据
```

这样的语法乍一看反复无常，实际上并非如此。如果 const 关键字出现在星号的左边，那么**指针所指向的**就是一个常量；如果 const 出现在星号的右边，那么**指针本身**就是一个常量 ;如果 const 同时出现在星号的两边, 那么两者就都是常量。

当指针所指的内容为常量时，一些程序员喜欢把 const 放在类型之前，其他一些人则喜欢放在类型后边，但要在星号的前边。这两种做法没有什么本质的区别，所以下边给出的两个函数声明的参数类型实际上是相同的：

```
void f1(const Widget *pw);          // f1 传入一个指向 Widget 对象常量的指针
void f2(Widget const *pw);          // f2 也一样
```

由于这两种形式在实际代码中都会遇到，所以二者你都要适应。

STL 迭代器是依照指针模型创建的，所以说 iterator 更像一个 T*指针。把一个 iterator 声明为 const 的更像是声明一个 const 指针（也就是声明一个 T* const 指针）：iterator 不允许指向不同类型的内容，但是其所指向的内容可以被修改。如果你希望一个迭代器指向某些不能被修改的内容（也就是指向 const T*的指针），此时你需要一个 const_iterator：

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter = vec.begin();
                                    // iter 就像一个 T* const
*iter = 10;                         // 正确，可以改变 iter 所指向的内容
++iter;                             // 出错！Iter 是一个 const

std::vector<int>::const_iterator cIter = vec.begin();
                                    // cIter 就像一个 const T*
*cIter = 10;                        // 出错！*cIter 是一个 const
++cIter;                            // 正确，可以改变 cIter
```

const 在函数声明方面还有一些强大的用途。在一个函数声明的内部，const 可以应用在返回值、单个参数，对于成员函数，可以将其本身声明为 const 的。

让函数返回一个常量通常可以在兼顾安全和效率问题的同时，减少客户产生错误的可能。好比有理数乘法函数（operator*）的声明，更多信息请参见条目 24。

```
class Rational { ... };
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

很多程序员在初次到这样的代码时都不会正眼看一下。为什么 operator*要返回一个 const 对象呢？这是因为如果不是这样，客户端将会遇到一些不愉快的状况，比如：

```
Rational a, b, c;
...
(a * b) = c;                    // 调用 operator= 能为 a*b 的结果赋值！
```

我不知道为什么一些程序员会企图为两个数的乘积赋值，但是我确实知道好多程序员的初衷并非如此。他们也许仅仅在录入的时候出了个小差错（他们的本意也许是一个布尔型的表达式）：

```
if (a * b = c) ...              // 啊哦…本来是想进行一次比较！
```

如果 a 和 b 是内建数据类型，那么这样的代码很明显就是非法的。避免与内建数据类型不必要的冲突，这是一个优秀的用户自定义类型的设计标准之一（另请参见条目 18），而允许为两数乘积赋值这让人看上去就很不必要。如果将 operator* 的返回值声明为 const 的则可以避免这一冲突，这便是要这样做的原因所在。

const 参数没有什么特别新鲜的——它与局部 const 对象的行为基本一致，你在应该在必要的时候尽可能使用它们。除非你需要更改某个参数或者局部对象，其余的所有情况都应声明为 const。这仅仅需要你多打六个字母，但是它可以使你从恼人的错误（比如我们刚才见到的“我本想打’==’但是却打了’=’”）中解放出来。

const 成员函数

对成员函数使用 const 的目的是：指明哪些成员函数可以被 const 对象调用。这一类成员函数在两层意义上是十分重要的。首先，它们使得类的接口更加易于理解。

很有必要了解哪些函数可以对对象做出修改而哪些不可以。其次，它们的出现使得与 const 对象协同工作成为可能。这对于高效编码来说是十分关键的一个因素，这是由于(将在条目 20 中展开解释)提高 C++程序性能的一条最基本的途径就是：传递对象的 const 引用。使用这一技术需要一个前提：就是必须要有 const 成员函数存在，只有它们能够处理随之生成的 const 对象。

如果若干成员函数之间的区别**仅仅**为“是否是 const 的”，那么它们也可以被重载。很多人都忽略了这一点，但是这是 C++的一个重要特征。请观察下面的代码，这是一个文字块的类：

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const
  { return text[position]; }       // operator[] ：用于 const 对象

  char& operator[](std::size_t position)
  { return text[position]; }       // operator[] ：用于非 const 对象

private:
   std::string text;
};
```

TextBlock 的 operator[]可以这样使用：

```
TextBlock tb("Hello");
std::cout << tb[0];               // 调用非 const 的 TextBlock::operator[]

const TextBlock ctb("World");
std::cout << ctb[0];              // 调用 const 的 TextBlock::operator[]
```

顺便说一下，在真实的程序中，const 对象在大多数情况下都以“传递指针”或“传递 const 引用”的形式出现。 上面的 ctb 的例子纯粹是人为的，而下面的例子在真实状况中常会出现：

```
void print(const TextBlock& ctb)   // 在这个函数中 ctb 是 const 的
{
```

```
  std::cout << ctb[0];              // 调用 const 的 TextBlock::operator[]
  ...
}
```

通过对 operator[]的重载以及为每个版本提供不同类型的返回值，你便可以以不同的方式处理 const 的或者非 const 的 TextBlock：

```
std::cout << tb[0];               // 正确：读入一个非 const 的 TextBlock
tb[0] = 'x';                      // 正确：改写一个非 const 的 TextBlock
std::cout << ctb[0];              // 正确：读入一个 const 的 TextBlock
ctb[0] = 'x';                     // 错误！不能改写 const 的 TextBlock
```

请注意，这一错误只与所调用的 operator[]的**返回值的类型**有关，然而对 operator[]调用本身的过程则不会出现任何问题。错误出现在企图为一个 const char&赋值时，这是因为 const char&是 operator[]的 const 版本的返回值类型。

同时还要注意的是，非 const 的 operator[]的返回值是一个 char 的**引用**，而不是 char 本身。如果 operator[]真的简单的返回一个 char，那么下面的语句将不能正确编译：

```
tb[0] = 'x';
```

这是因为，企图修改一个返回内建数据类型的函数的返回值根本都是非法的。即使假设这样做合法，而 C++是通过传值返回对象的，所修改的仅仅是由 tb.text[0]复制出的一份**副本**，而不是 tb.text[0]本身，你也不会得到预期的效果。

让我们暂停一小会儿，来考虑一下这里边的哲学问题。把一个成员函数声明为 const 的有什么涵义呢？这里有两个流行的说法：**按位恒定**（也可叫做**物理恒定**）和**逻辑恒定**。

按位恒定阵营坚信：当且仅当一个成员函数对于其所在对象所有的数据成员（static 数据成员除外）都不做出改动时，才需要将这一成员函数声明为 const，换句话说，将成员函数声明为 const 的条件是：成员函数不对其所在对象内部做任何的改动。按位恒定有这样一个好处，它使得对违反规则行为的检查十分轻松：编译器仅需要查找对数据成员的赋值操作。实际上，按位恒定就是 C++对于恒定

的定义，如果一个对象调用了某个 const 成员函数，那么该成员函数对这个对象内所有非静态数据成员的修改都是不允许的。

不幸的是，大多数不完全 const 的成员函数也可以通过按位恒定的测试。在特定的情况下，如果一个成员函数频繁的修改一个指针**所指的位置**，那么它就不应是一个 const 成员函数。但是只要这个指针存在于对象内部，这个函数就是按位恒定的，这时候编译器不会报错。这样会导致违背常理的行为。比如说，我们手头有一个类似于 TextBlock 的类，其中保存着 char*类型的数据而不是 string，因为这段代码有可能要与一些 C 语言的 API 交互，但是 C 语言中没有 string 对象一说。

```
class CTextBlock {
public:
  ...
  char& operator[](std::size_t position) const
  // operator[]不恰当的（但是符合按位恒定规则）定义方法
  { return pText[position]; }

private:
  char *pText;
};
```

尽管 operator[]返回一个对象内部数据的引用，这个类仍（不恰当地）将其声明为 const 成员函数（条目 28 将深入讨论这个问题）。先忽略这个问题，请注意这里的 operator[]实现中并没有以任何形式修改 pText。于是编译器便会欣然接受这样的做法，毕竟，它是按位恒定的，所有的编译器所检查的都是这一点。但是请观察，在编译器的纵容下，还会有什么样的事情发生：

```
const CTextBlock cctb("Hello");    // 声明常量对象

char *pc = &cctb[0];               // 调用 const 的 operator[]
                                   // 从而得到一个指向 cctb 中数据的指针

*pc = 'J';                         // cctb 现在的值为"Jello"
```

当你创建了一个包含具体值的对象常量后，你仅仅通过对其调用 const 的成员函数，就可以改变它的值！这显然是有问题的。

逻辑恒定应运而生。坚持这一宗旨的人们争论到：如果某个对象调用了一个 const 的成员函数，那么这个成员函数可以对这个对象内部做出改动，但是仅仅以客户端无法察觉的方式进行。比如说，你的 CTextBlock 类可能需要保存文字块的长度，以便在需要的时候调用：

```
class CTextBlock {
public:
  ...
  std::size_t length() const;

private:
  char *pText;
  std::size_t textLength;              // 最后一次计算出的文字块长度
  bool lengthIsValid;                  // 当前长度是否可用
};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);  // 错误！不能在 const 成员函数中
    lengthIsValid = true;             // 对 textLength 和 lengthIsValid 赋值
  }
  return textLength;
}
```

以上 length 的实现绝不是按位恒定的。这是因为 textLength 和 lengthIsValid 都可以改动。尽管看上去它应该对于 CTextBlock 对象常量可用，但是编译器会拒绝。编译器始终坚持遵守按位恒定。那么该怎么办呢？

解决方法很简单：利用 C++中与 const 相关的灵活性，使用可变的（mutable）数据成员。mutable 可以使非静态数据成员不受按位恒定规则的约束：

```
class CTextBlock {
public:
```

```
  ...
  std::size_t length() const;

private:
  char *pText;

  mutable std::size_t textLength;          // 这些数据成员在任何情况下均可修改
  mutable bool lengthIsValid;              // 在 const 成员函数中也可以
};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);       // 现在可以修改了
    lengthIsValid = true;                  // 同上
  }
  return textLength;
}
```

避免 const 与非 const 成员函数之间的重复

mutable 对于“我不了解按位恒定”的情况不失为一个良好的解决方案，但是它并不能对于所有的 const 难题做到一劳永逸。举例说，TextBlock（以及 CTextBlock）中的 operator[]不仅仅返回一个对恰当字符的引用，同时还要进行边界检查、记录访问信息，甚至还要进行数据完整性检测。如果将所有这些统统放在 const 或非 const 函数（我们现在会得到过于冗长的隐式内联函数，不过不要惊慌，在条目 30 中这个问题会得到解决）中，看看我们会得到什么样的庞然大物：

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const
  {
    ...                                    // 边界检查
    ...                                    // 记录数据访问信息
```

```
    ...                                // 确认数据完整性
    return text[position];
  }
  char& operator[](std::size_t position)
  {
    ...                                // 边界检查
    ...                                // 记录数据访问信息
    ...                                // 确认数据完整性
    return text[position];
  }

private:
   std::string text;
};
```

啊哦！重复代码让人头疼，还有随之而来的编译时间增长、维护成本增加、代码膨胀，等等……当然，像边界检查这一类代码是可以移走的，它们可以单独放在一个成员函数（很自然是私有的）中，然后让这两个版本的 operator[]来调用它，但是你的代码仍然有重复的函数调用，以及重复的 return 语句。

对于 operator[]你真正需要的是：一次实现，两次使用。也就是说，你需要一个版本的 operator[]来调用另一个。这样便可以通过转型来消去函数的恒定性。

通常情况下转型是一个坏主意，后边我将专门用一条来告诉你为什么不要使用转型（条目 21），但是代码重复也不会让人感到有多轻松。在这种情况下，const 版的 operator[]与非 const 版的 operator[]所做的事情完全相同，不同的仅仅是它的返回值是 const 的。通过转型来消去返回值的恒定性是安全的，这是因为任何人调用这一非 const 的 operator[]首先必须拥有一个非 const 的对象，否则它就不能调用非 const 函数。所以尽管需要一次转型，在 const 的 operator[]中调用非 const 版本，可以安全地避免代码重复。下面是实例代码，读完后边的文字解说你会更明了。

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const
```

```
  {                                   // 同上
    ...
    ...
    ...
    return text[position];
  }
  char& operator[](std::size_t position)
  {                                   // 现在仅调用 const 的 op[]
    return
      const_cast<char&>(              // 消去 op[]返回值的 const 属性
        static_cast<const TextBlock&>(*this)
                                      // 为*this 的类型添加 const 属性；
        [position];                   // 调用 const 版本的 op[]
      );
  }
  ...
};
```

就像你所看到的，上面的代码进行了两次转型，而不是一次。我们要让非 const 的 operator[]去调用 const 版本的，但是如果在非 const 的 operator[]的内部，我们只调用 operator[]而不标明 const，那么函数将对自己进行递归调用。那将是成千上万次的毫无意义的操作。为了避免无穷递归的出现，我们必须要指明我们要调用的是 const 版本的 operator[]，但是手头并没有直接的办法。我们可以用*this 从其原有的 TextBlock&转型到 const TextBlock&。是的，我们使用了一次转型添加了一个 const！这样我们就进行了两次转型：一次为 *this 添加了 const（于是对于 operator[]的调用将会正确地选择 const 版本），第二次转型消去了 const operator[]返回值中的 const。

添加 const 属性的那次转型是为了强制保证转换工作的安全性（从一个非 const 对象转换为一个 const 对象），我们使用 static_cast 来进行。消去 const 的那次转型只可以通过 const_cast 来完成，所以这里实际上也没有其他的选择。（从技术上讲还是有的。C 语言风格的转型在这里也能工作，但是，就像我在条目 27 中所解释的，这一类转型在很多情况下都不是好的选择。如果你对于 static_cast 和 const_cast 还不熟悉，条目 27 中有相关介绍。）

在众多的示例中，我们最终选择了一个运算符来进行演示，因此上面的语法显得有些古怪。这些代码可能不会赢得任何选美比赛，但是通过以 const 版本的形式实现非 const 版本的 operator[]，可以避免代码重复，这正是我们所期望的效果。为达到这一目标而写下看似笨拙的代码，这样做是否值得全看你的选择，但是，以 const 版本的形式来实现非 const 的成员函数——了解这一技术肯定是值得的。

更值得你了解的是，上面的操作是**不可逆**的，即：通过让 const 版本的函数调用非 const 版本来避免代码重复，是不可行的。请记住，一个 const 成员函数应保证永远不会更改其所在对象的逻辑状态，但是一个非 const 的成员函数无法做出这样的保证。如果你在一个 const 函数中调用了一个非 const 函数，那么你曾保证不会被改动的对象就有被修改的风险。这就是为什么说让一个 const 成员函数调用一个非 const 成员函数是错误的：对象有可能被修改。实际上，为了使代码能够得到编译，你还需要使用一个 const_cast 来消去*this 的 const 属性，显然这是不必要的麻烦。上一段中按相反的调用次序才是安全的：非 const 成员函数可以对一个对象做任何想做的事情，因此调用一个 const 成员函数不会带来任何风险。这就是为什么 static_cast 可以这样操作*this 的原因：这里不存在 const 相关的危险。

就像本条目一开始所说的，const 是一个令人赞叹的东西。对于指针和迭代器，以及对于指针、迭代器和引用所涉及的对象，对于函数的参数和返回值，对于局部变量，以及对于成员函数来说，const 都是一个强大的伙伴。尽可能去利用它。你一定不会后悔。

时刻牢记

- 将一些东西声明为 const 可以帮助编译器及时发现用法上的错误。const 可以用于各个领域，包括任意作用域的对象、函数参数和返回值、成员函数。
- 编译器严格遵守按位恒定规则，但是你应该在需要时应用逻辑恒定。
- 当 const 和非 const 成员函数的实现在本质上相同时，可以通过使用一个非 const 版本来调用 const 版本来避免代码重复。

条目4： 确保对象在使用前得到初始化

C++在对象值的初始化问题上显得变幻莫测。比如说，你写下了下面的代码：

```
int x;
```

在一些上下文里，x 会确保得到初始化（为零），但是另一些情况下则不会，如果你这样编写：

```
class Point {
  int x, y;
};
...
Point p;
```

p 的数据成员在一些情况下会确保得到初始化（为零），但是另一些情况则不会。如果你以前学习的语言没有对象初始化的概念，那么请你注意了，因为这很重要。

读取未初始化的数据将导致未定义行为。在一些语言平台中，通常情况下读取未初始化的数据仅仅是使你的程序无法运行罢了。更典型的情况是，这样的读取操作可能会得到内存中某些位置上的半随机的数据，这些数据将会“污染”需要赋值的对象，最终，程序的行为将变得十分令人费解，你也会陷入烦人的除错工作中。

现在，人们制定了规则来规定对象在什么时候必须被初始化，以及什么时候不会。但是遗憾的是，这些规则太过复杂了——在我看来，你根本没必要去记忆它们。整体上讲，如果你正在使用 C++中 C 语言的一部分（参见条目 1），并且这里的初始化会引入运行时开销，那么此时初始化工作无法确保完成。但当你使用非 C 的 C++部分时，情况有时就会改变。这便可以解释为什么数组（C++中的 C 语言）不会确保得到初始化，而一个 vector（C++中的 STL）会。

解决这类表面上的不确定性问题最好的途径就是 :总是在使用对象之前对它们进行初始化。对于内建类型的非成员对象，你需要手动完成这一工作。请看下边的示例：

```
int x = 0;                                  // 手动初始化一个 int 值
const char * text = "A C-style string";     // 手动初始化一个指针（见条目 3）
double d;
std::cin >> d;                              // 通过读取输入流进行“初始化”
```

对于其他大多数情况而言，初始化的重担就落在了构造函数的肩上。这里的规则很简单：确保所有构造函数都对整个对象做出完整的初始化。

遵守这一规则是件很容易的事情，但是还有件重要的事：不要把赋值和初始化搞混了。请看下边示例中的构造函数，它是通讯录中用于表示条目的类：

```
class PhoneNumber { ... };

class ABEntry {                     // ABEntry = "Address Book Entry"
public:
  ABEntry(const std::string& name, const std::string& address,
          const std::list<PhoneNumber>& phones);
private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int num TimesConsulted;
};

ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
{
  theName = name;                   // 以下这些是赋值，而不是初始化
  theAddress = address;
  thePhones = phones;
  numTimesConsulted = 0;
}
```

上边的做法可以让你得到一个包含你所期望值的 ABEntry 对象，但是这仍不是最优的做法。C++的规则约定：一个对象的数据成员要在进入构造函数内部**之前**得到初始化。在 ABEntry 的构造函数内部，theName、theAddress 以及 thePhones 并不是得到了初始化，而是**被赋值**了。初始化工作应该在更早的时候进行：在进入 ABEntry 构造函数内部之前，这些数据成员的默认构造函数应该自动得到调用。注意这对于 numTimesConsulted 不成立，因为它是内建数据类型的。对它而言，在被赋值以前，谁也不能确保它得到了初始化。

编写 ABEntry 的构造函数的一个更好的办法是使用成员初始化表，而不是为成员一一赋值：

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
: theName(name),
  theAddress(address),              // 现在这些是初始化
  thePhones(phones),
  numTimesConsulted(0)
{}                                  // 现在构造函数内部是空的
```

如果仅看运行结果，上面的构造函数与更靠前一些的那个是一样的，但是后者的效率更高些。为数据成员赋值的版本首先调用了 theName、theAddress 以及 thePhones 的默认构造函数来初始化它们，在默认构造函数已经为它们分配好了值之后，立即又为它们重新赋了一遍值。于是默认构造函数的所有工作就都白费了。使用成员初始化表的方法可以避免这一浪费，这是因为：初始化表中的参数对于各种数据成员均使用构造函数参数的形式出现。这样，theName 就通过复制 name 的值完成了构造，theAddress 通过复制 address 的值完成构造，thePhones 通过复制 phones 的值完成构造。对于大多数类型来说，相对于“先调用默认构造函数再调用拷贝运算符”而言，通过单一的调用拷贝构造函数更加高效——在一些情况下**尤其**明显。

对于内建类型的对象，比如 numTimeConsulted，初始化与赋值的开销是完全相同的，但是为了保证程序的一致性，最好通过成员初始化的方式对所有成员进行初始化。类似地，即使你期望让默认构造函数来构造一个数据成员，你仍可以使用成员初始化表，只是不为初始化参数指定一个具体的值而已。比如，如果 ABEntry 拥有一个没有参数的构造函数，它可以这样实现：

```
ABEntry::ABEntry()
:theName(),                         // 调用 theName 的默认构造函数；
 theAddress(),                      // theAddress 和 thePhones 同上；
 thePhones(),                       // 但是 numTimesConsulted
 numTimesConsulted(0)               // 一定要显性初始化为零
 {}
```

由于成员初始化表中没有为用户定义类型的数据成员指定初始值时，编译器会自动为这些成员调用默认构造函数，因此一些程序员会认为上文中的做法显得有些多余。这是可以理解的，但是“总将每个数据成员列在初始化表中”这一策略可以避免你去回忆列表中是哪个成员被忽略了从而无法得到初始化。比如说，如果你因为 numTimesConsulted 是内建数据类型的，就不将其列入成员初始化表中，那么你的代码便极有可能呈现出未定义行为。

有些时候使用初始化表是**必须的**，即使是对于内建类型。举例说，const 或者引用的数据成员必须得到初始化，它们不能被赋值（另请参看条目 5）。至于数据成员什么时候必须在成员初始化表中进行初始化，什么时候没有必要，如果你不希望去记忆这些规则，那么最简便的选择就是**永远**都使用初始化表。一些时候初始化表是必须的，而且通常会获得比赋值更高的效率。

许多类都包含多个构造函数，每个构造函数都有自己的成员初始化表。如果某个类拥有非常多的数据成员和/或基类时，这些初始化列表中将会存在不少无意义的重复代码，程序员们也会感到厌烦。在这种情况下，忽略表中的一些条目也并非毫无意义，这些忽略的数据成员应符合这一条件：对它们进行赋值还是真正的初始化没有什么差别。可以把这些赋值语句放在一个单一（当然是私有的）的函数里，并让所有的构造函数在必要的时候调用这个函数。在数据成员要接收的真实的初始化数据需要从某个文件中读取时，或者要到某个数据库中去查找时，这一方法尤其有用。但是总体而言，真正的成员初始化终究要比通过赋值进行伪初始化要好。

C++也不是总那么变幻莫测，对象中数据的初始化的顺序就是 C++的稳定因素之一。这个次序通常情况下是一致的：基类应在派生类之前得到初始化（另参见条目 12），在类的内部，数据成员应以它们声明的顺序得到初始化。比如说在 ABEntry 内部，theName 永远都是第一个得到初始化的，theAddress 第二，thePhones 第三，numTimesConsulted 最后。即使它们在成员初始化表中的排列顺序不同于声明次序，（尽管这样做看上去应该算作非法，但不幸的是事实并非这样。）上述初始化顺序也会得到遵循。为了不使读者陷入困惑，也为了避免日后出现让人难以理解的 bug，你应该保证初始化表中成员的顺序与它们被声明时的顺序严格一致。

在你完成了对内建类型的非成员对象的显式初始化，并且确保了构造函数使用成员初始化表对基类和数据成员进行了初始化之后，需要你关心的内容就仅剩下了一个，那就是（先长舒一口气）：在不同的置换单元中，非局部静态对象的初始化次序是怎样的。

让我们来抽丝剥茧分析这个问题：

【静态对象（static object）】一个静态对象在被构造之后，它的寿命一直延续到程序结束。保存在栈或堆中的对象都不是这样。静态对象包括：全局对象、名字空间域对象、类内部的 `static` 对象、函数内部的 `static` 对象，文件域的 `static` 对象。函数内部的静态对象通常叫做**局部静态对象**（这是因为它们对于函数而言是局部的），其它类型的静态对象称为**非局部静态对象**。静态对象在程序退出的时候会被自动销毁，换句话说，在 `main` 中止运行的时候，静态对象的析构函数会自动得到调用。

【置换单元（translation unit）】一个置换单元是这样一段源代码：由它可以生成一个目标文件。总的来说置换单元就是单独一个代码文件，以及所有被`#include`进来的文件。

于是，我们所要解决的问题中，至少包含两个需要单独编译的源码文件，每一个都至少包含一个非局部静态对象（换句话说，是一个全局的，或者名字空间域的，或类内部或者文件域的 `static` 对象）。真正的问题是：如果一个置换单元内的一个非局部静态对象的初始化工作利用了另一个置换空间内的另一个非局部静态变量，那么所使用的对象应该是未经初始化的，这是因为：**定义在不同置换单元内的非静态对象的初始化工作的顺序是未定义的**。

这里一个示例可以帮助我们理解这一问题。假设你编写了一个 `FileSystem` 类，它可以让 Internet 上的文件看上去像是本地的。由于你的类要使得整个世界看上去像是一个单一的文件系统，你应该创建一个专门的类来代表这个单一的文件系统，让这个类拥有全局的或者名字空间的作用域：

```
class FileSystem {                  // 来自你的库
public:
  ...
  std::size_t numDisks() const;     // 许多成员函数中的一个
```

```
  ...
};

extern FileSystem tfs;              // 供客户端使用的对象
                                    // "tfs" = "the file system"
```

一个 FileSystem 对象绝对是重量级的，所以说在 tfs 对象被构造之前使用它会带来灾难性后果。

现在设想一下，一些客户为文件系统创建了一个文件夹的类。很自然地，他们的类会使用 tfs 对象。

```
class Directory {                       // 由类库的客户创建
public:
   Directory( params );
  ...
};

Directory::Directory( params )
{
  ...
  std::size_t disks = tfs.numDisks();  // 使用 tfs 对象
  ...
}
```

进一步设想，客户可能会为临时文件创建一个单独的 Directory 对象：

```
Directory tempDir( params );        // 存放临时文件的文件夹
```

现在，初始化次序的重要性已然浮出水面：除非 tfs 在 tempDir 初始化之前得到初始化，否则 tempDir 的构造函数将会尝试在 tfs 被初始化之前使用它。但是 tfs 和 tempDir 是由不同的人、在不同的时间、在不同的源码文件中创建的——这两者都是非局部静态对象，它们定义于不同的置换单元中。那么你如何保证 tfs 在 tempDir 之前得到初始化呢？

事实上这是不可能的。重申一遍，**定义在不同置换单元内的非静态对象的初始化工作的顺序是未定义的**。当然这是有理由的：为非局部静态对象确定“恰当的”初始化

顺序是一件很有难度的工作。非常有难度。难到根本无法解决。在其大多数形式——由隐式模板实例化产生的多个置换单元和非局部静态对象（也许它们是通过隐式模板实例化自行生成的）——这不仅使得确认初始化的顺序变得不可能，甚至寻找一种可行的初始化顺序的特殊情况，都显得毫无意义。

幸运的是，一个小小的方法可以完美的解决这个难题。所要做的仅仅是把每个非局部静态对象移入为它创建的专用函数中，函数要声明为 `static` 的。这些函数返回一个它们所属对象的引用。于是客户就可以调用这些函数，而不是直接使用那些对象。也就是说，非局部静态对象被**局部**静态对象取代了。（设计模式迷们很容易发现，这是单例模式（Singleton Pattern）一个通用实现。）

这一方法基于 C++的一个约定，那就是：对于局部静态对象来说，在其被上述函数调用的时候，程序中第一次引入了该对象的定义，它在此时就一定会得到初始化。所以如果你不去直接访问非局部静态对象，而改用“通过函数返回的引用来调用局部静态对象”，那么你就保证了你得到的这一引用将指向一个已经初始化的对象。作为奖励，如果你从未调用过模仿非局部静态对象的函数，你的程序就永远不会引入对这类对象进行构造和析构的开销，而这对于真正的非局部静态对象来说是不可能的。

下面是关于 `tfs` 和 `tempDir` 对这一技术的应用：

```
class FileSystem { ... };           // 同上

FileSystem& tfs()                   // 这一函数代替了 tfs 对象；它在
                                    // FileSystem 类中应该是 static 的
{
  static FileSystem fs;             // 对局部静态对象的定义和初始化
  return fs;                        // 返回该对象的引用
}

class Directory { ... };            // 同上

Directory::Directory( params )      // 同上，但对 tfs 的引用现在为对 tfs()
{
  ...
```

```
  std::size_t disks = tfs().numDisks();
  ...
}

Directory& tempDir()                    // 这个函数取代了 tempDir 对象；它在
                                        // Directory 类中应该是 static 的
{
  static Directory td;                  // 对局部静态对象的定义和初始化
  return td;                            // 返回该对象的引用
}
```

这一改进系统不需要客户做出任何改变，除了他们所引用的是tfs()和tempDir()而不是 tfs 和 tempDir。也就是说，他们使用的是返回引用的函数而不是直接使用对象本身。

编写这一类返回引用的函数所需要遵循的方针总是十分简单的：第 1 行定义和初始化一个局部静态对象，第 2 行返回它的引用。如此的简单易用使得这类函数非常适合作为内联函数，尤其是对它们的调用非常频繁时（参见条目 30）。另外，这些函数中包含着静态对象，这一事实使得他们在多线程系统中也会遇到问题。在此声明，任何种类的非 const 静态对象，无论是局部的还是非局部的，它们面对多线程都会碰到这样那样的问题。解决这一问题的方法之一是：在程序还以单线程状态运行时，手动调用所有的这类返回引用的函数。这可以排除与初始化相关的竞争状态的出现。

当然，使用此类返回引用的函数来防止初始化次序问题的理念，首先基于此处存在一个合理的初始化次序。如果你的系统要求对象 A 必须在对象 B 之前得到初始化，但是 A 的初始化需要以 B 的初始化为前提，你将会面临一个问题，坦白说，你是咎由自取。然而，如果你能够驾驭这一不正常的境况，这里介绍的解决方法仍然可以良好的为你服务，至少对于单线程应用程序来说是这样的。

为了避免在对象初始化之前使用它，你仅仅需要做三件事。第一，手动初始化内建类型的非成员对象。第二，使用成员初始化表来初始化对象的每一部分。最后，初始化顺序的不确定性使得定义于不同置换空间里非局部静态对象难以正常运行，你需要寻求一个新的设计方案。

时刻牢记

- 由于 C++只在某些情况下对于内建类型对象进行初始化，所以对它们要进行手动初始化。
- 对于构造函数，要尽量使用成员初始化表，避免在构造函数内部进行复制。初始化表中的次序要与成员在类中被声明的次序相一致。
- 要避免跨置换单元的初始化次序问题发生，可以使用局部静态对象来代替非局部静态对象的方案来解决。

第二章. 构造函数、析构函数、赋值运算符

几乎你编写的每个类都有一个或多个构造函数、一个析构函数、和一个拷贝赋值运算符。这没有什么好稀奇的。这些是编写一个类所必需的一些函数，这些函数控制着类的基本操作，其中包括使一个对象由概念变为现实并且确保这一对象得到初始化，以及从系统中排除一个对象并对其进行恰当的清理工作，还有为一个对象赋予一个新值。在这些函数中出错将为你的类带来深远而重大的负面影响，这自然是令人扫兴的，所以写好这些函数是十分重要的。这些函数构成了类的中枢神经，这一章中将为你介绍怎样编写这些程序才会使你的类更加优秀。

条目5： 要清楚 C++在后台为你书写和调用了什么函数

什么时候一个空类在实际上并不是空类呢？我们说，在 C++处理它的时候。对于一个类来说，如果你不自己手动声明一个复制构造函数、一个赋值运算符、和一个析构函数，编译器就会自动为你声明这些函数。而且，如果你根本没有声明构造函数的话，编译器也将为你声明一个默认构造函数。所有这些函数将是 `public` 的并且是 `inline` 的（参见条目 30）。举例说，如果你编写了：

```
class Empty{};
```

它在本质上讲与下边这个类是等价的：

```
class Empty {
public:
  Empty() { ... }                       // 默认构造函数
  Empty(const Empty& rhs) { ... }       // 拷贝构造函数
  ~Empty() { ... }                      // 析构函数，下文将分析它是否为虚函数
  Empty& operator=(const Empty& rhs)    // 赋值运算符
  { ... }
};
```

这些函数只有在需要的时候才会生成，但是需要他们是经常的事情。以下的代码可以生成每一个函数：

```
Empty e1;                       // 默认构造函数
```

```
                                    // 析构函数
Empty e2(e1);                       // 复制构造函数
e2 = e1;                            // 赋值运算符
```

现在我们知道编译器为你编写了这些函数，那么这些函数是做什么的呢？默认构造函数和析构函数主要作用是为编译器提供一个放置“幕后代码”的空间，“幕后代码”完成的是诸如对于基类和非静态数据成员的构造函数和析构函数的调用。请注意，对于由编译器生成的析构函数，除非所在的类继承自一个拥有虚析构函数的基类（这个情况下，析构函数的虚拟性继承自它的基类），其他情况均不是虚函数（参见条目 7）。

对于复制构造函数和赋值运算符而言，编译器所生成的版本只是简单地把原对象中所有的非静态数据成员复制到目标对象里。请参见下边的 `NamedObject` 模板，它让你能够使用名字来访问 `T` 类型对象：

```
template<typename T>
class NamedObject {
public:
  NamedObject(const char *name, const T& value);
  NamedObject(const std::string& name, const T& value);
  ...
private:
  std::string nameValue;
  T objectValue;
};
```

由于 `NamedObject` 中声明了一个构造函数，编译器则不会为你自动生成一个默认构造函数。这一点很重要。这意味着如果这个类已经经过你认真仔细的设计，你认为它的构造函数必须包含参数，这时候你便不需要担心编译器会违背你的意愿，轻率地在你的类中添加一个没有参数的构造函数。

`NamedObject` 没有声明复制构造函数和赋值运算符，所以编译器将会自动生成这些函数（在需要的时候）。请看下面代码中对复制构造函数的应用：

```
NamedObject<int> no1("Smallest Prime Number", 2);
NamedObject<int> no2(no1);          // 调用复制构造函数
```

由编译器自动生成的这一复制构造函数必须要分别使用 no1.nameValue 和 no1.objectValue 来初始化 no2.nameVaule 和 no2.objectValue。nameValue 是一个 string，由于标准字符串类型带有一个复制构造函数，所以 no2.nameValue 将通过调用 string 的复制构造函数（以 no1.nameValue 作为其参数）得到初始化。另外，NamedObject<int>::ObjectValue 是 int 型的（这是因为对于当前的模板实例来说，T 是 int 型的），而 int 是一个内建类型，所以 no2.objectValue 将通过复制 no1.objectValue 来得到初始化。

由编译器自动生成的 NamedObject<int>的拷贝赋值运算符与上述复制构造函数的行为基本一致，但是大体上讲，编译器自动生成的拷贝赋值运算符只有在生成代码合法、有存在的价值时，才会像上文中我所描述的那种方式运行。如果其中任意一条无法满足，编译器将会拒绝为你的类生成一个 operator=。

请看下边的示例，如果 NamedObject 被定义成这样，nameValue 是一个指向字符串的**引用**，而 objectValue 是一个 **const** T：

```
template<class T>
class NamedObject {
public:
  // 以下的构造函数中的 name 参数不再是 const 的了，
  // 这是因为现在 nameValue 是一个指向非 const 的 string 的引用。
  // char*参数的构造函数已经不复存在了，
  // 这是因为这里必须存在一个 string 引用。
  NamedObject(std::string& name, const T& value);
  ...                                  // 如前所述，假设没有声明任何 operator=

private:
  std::string& nameValue;              // 现在是一个引用
  const T objectValue;                 // 现在为 const 的
};
```

现在请你思考接下来会发生什么事情：

```
std::string newDog("Persephone");
std::string oldDog("Satch");
```

```
NamedObject<int> p(newDog, 2);      // 在我最初编写这段代码的时候我们的狗
                                    // Persephone 正要度过她的两周岁生日

NamedObject<int> s(oldDog, 36);     // 我家的狗 Satch（在我小时候养的）
                                    // 如果她现在还活着应该有 36 岁了

p = s;                              // 对于 p 中的数据成员将会发生什么呢？
```

在赋值之前，p.nameValue 和 s.nameValue 都引用了一个 string 对象，但不是同一个。那么赋值操作又怎么会影响到 p.nameValue 呢？在赋值之后，p.nameValue 是否应引用由 s.nameValue 中保存的 string 应用呢？换句话说，引用是否可以被更改呢？如果可以的话，我们就开创了一个全新的议题，因为 C++并没有提供任何方法让一个引用改变其所引用的对象。换个角度说，p.nameValue 所引用的 string 对象是否可以被修改，从而影响到包含指向这个 string 对象的指针或引用的其他对象（换句话说，此次赋值中未直接涉及到的对象）呢？这是否是编译器自动生成的拷贝赋值运算符应该做的呢？

面对这一难题，C++拒绝编译这类代码。如果你希望让包含引用成员的类支持赋值操作，就必须自己定义拷贝赋值运算符。对于包含 const 成员的类（比如上文中修改后的 objectValue）编译器也会做类似处理。由于修改 const 成员是非法的，因此编译器无法在一个隐式生成的赋值函数中确定如何处理它们。最终，如果一个基类中的拷贝赋值运算符是声明为 private 的，那么在派生类中编译器将拒绝隐式生成拷贝赋值运算符。毕竟，编译器为派生类自动生成的拷贝赋值运算符也要处理基类中相应的部分（参见条目 12），但是在这些拷贝赋值运算符处理相应的基类部分时，是肯定不能调用派生类中无权调用的数据成员的。

时刻牢记

- 编译器可能会隐式为一个类生成默认构造函数、复制构造函数、拷贝赋值运算符和析构函数。

条目6： 要显式禁止编译器为你生成不必要的函数

真实的房地产代理商的工作是出售房屋，而一个为房产代理商提供支持的软件系统自然要用一个类来代表要出售的房屋：

```
class HomeForSale { ... };
```

所有房地产代理商能够很轻松的指出，每套房产都是独一无二的——没有两套是完全一样的。既然如此，为一个 HomeForSale 对象复制出一个**副本**的想法就显得没什么意义了。你怎么能够复制那些生来就独一无二的东西呢？如果你尝试去复制一个 HomeForSale 对象，那么编译器则不应该接受：

```
HomeForSale h1;
HomeForSale h2;

HomeForSale h3(h1);                    // 尝试复制 h1：不应通过编译！
h1 = h2;                               // 尝试复制 h2：不应通过编译！
```

可惜的是，防止这种复杂问题发生的方法并不是那么直截了当。通常情况下，如果你希望一个类不支持某种特定的功能，你需要做的仅仅是不去声明那个函数。然而这一策略对复制构造函数和拷贝赋值运算符就失效了，这是因为，即使你不做声明，而一旦有人尝试调用这些函数，编译器就会为你自动声明它们（参见条目 5）。

这会使你便陷入困境。如果你不声明一个复制构造函数或者赋值运算符，编译器可能就会帮你去做，你的类就会支持对象复制。从另一个角度说，如果你确实声明了这些函数，你的类仍然支持复制。但是现在的目标是**防止**复制！

解决问题的关键是，所有编译器生成的函数都是公共的。为了防止编译器生成这些函数，你必须自己声明，但是现在没有什么要求**你**将这些函数声明为公共的。取而代之，你应该将复制构造函数和赋值运算符声明为**私有的**。通过显式声明一个函数，你就可以防止编译器去自动生成这个函数，同时，通过将函数声明为 private 的，你便可以防止人们去调用它。

差不多了。但是这一方案也没有那么傻瓜化，这是因为成员函数和友元函数仍然可以调用你的私有函数。**除非**你足够的聪明，没有去**定义**这些成员或友元。如果一些人由于疏忽大意而调用了其中的任意一个，他们会在程序连接时遇到一个错误。把

成员函数声明为 private 的但是不去实现它们，这一窍门已经成为编程常规，在C++的 I/O 流的库中的一些类，都会采用这种方法来防止复制。比如，你可以参考标准库中 ios_base、basic_ios 和 sentry 的实现。你会发现在各种情况下，复制构造函数和拷贝赋值运算符都声明为 private 而且没有得到定义。

对 HomeForSale 使用这一技巧十分简单：

```
class HomeForSale {
public:
  ...
private:
  ...
  HomeForSale(const HomeForSale&);          // 只有声明
  HomeForSale& operator=(const HomeForSale&);
};
```

你会发现我省略了函数参数的名称。这样做并不是必需的，这仅仅是一个普通惯例。毕竟这些代码不会得到实现，而且很少会用到，那么给这些参数命名又有什么用呢？

通过上文中类的声明，编译器会防止客户尝试复制 HomeForSale 对象，如果你不小心在成员函数或者友元函数中这样做了，那么你的程序将无法得到连接。

如果你将复制构造函数和拷贝赋值运算符声明为 private 的，并且将二者移出 HomeForSale 的内部，放置在一个专门设计用来防止复制的基类中，那么在编译时就排除这些原本是连接时的错误便成为可能（这是件好事——早期发现错误要比晚些更理想）。这一基类极其简单：

```
class Uncopyable {
protected:                              // 允许派生类存在构造函数和析构函数
  Uncopyable() {}
  ~Uncopyable() {}

private:
  Uncopyable(const Uncopyable&);   // 但禁止复制
  Uncopyable& operator=(const Uncopyable&);
};
```

为了防止 HomeForSale 对象被复制，我们所需要做的仅仅是让其继承 Uncopyable：

```
class HomeForSale: private Uncopyable {
  ...                                    // 这一类不再声明复制构造函数和赋值运算符
};
```

这样做是可行的，这是因为如果有人（甚至是一个成员或友元函数）尝试复制一个 HomeForSale 对象，编译器将会尝试自动生成一个复制构造函数和一个拷贝赋值运算符。就像条目 12 中所解释的，这些函数由编译器自动生成的版本会尝试调用它们基类中的这一部分，显然这些调用只能吃到闭门羹，这是因为复制操作在基类中是私有的。

Uncopyable 的实现和应用存在一些微妙的问题，诸如继承自 Uncopyable 的类不一定必须为 public 的（参见条目 32 和条目 39），Uncopyable 的析构函数不一定必须为虚函数（参见条目 7）。由于 Uncopyable 不包含任何数据，它适合作为空基类优化方案（参见条目 39），但是由于它是一个基类，使用这一技术将导致多重继承（参见条目 40）。然而，多重继承在某种情况下会使空基类优化失去作用（同样，请参见条目 39）。总体来说，你可以忽略这些微妙的问题，仅仅使用上文中的 Uncopyable，因为它会像所承诺的那样精确地完成工作。你也可以使用 Boost 版本（条目 55）。那个类叫做 noncopyable。它是一个优秀的类，我只是发现它的名字显得有些不自然（un-natural），呃，“非”自然（non-natural）。

时刻牢记

- 为了禁用编译器自动提供的功能，你必须将相关的成员函数声明为 private 的，同时不要实现它。方法之一是：使用一个类似于 Uncopyable 的基类。

条目7： 要把多态基类的析构函数声明为虚函数

现在考虑一个计时器的问题，我们首先创建一个名为 TimeKeeper 的基类，然后在它的基础上创建各种派生类，从而用不同手段来计时。由于计时有很多方式，所以这样做是值得的：

```
class TimeKeeper {
```

```
public:
  TimeKeeper();
  ~TimeKeeper();
  ...
};

class AtomicClock: public TimeKeeper { ... };  // 原子钟
class WaterClock: public TimeKeeper { ... };   // 水钟
class WristWatch: public TimeKeeper { ... };   // 腕表
```

许多客户希望在访问时间时不用关心计算的细节，所以在此可以使用一个**工厂函数**来返回一个指向计时器对象的指针，工厂函数会返回一个基类指针，这个指针将指向一个新创建的派生类对象：

```
TimeKeeper* getTimeKeeper();       // 返回一个继承自 TimeKeeper 的动态分配的对象
```

为了不破坏工厂函数的惯例，getTimeKeeper 返回的对象将被放置在堆上，所以必须要在适当的时候删除每一个返回的对象，从而避免内存或者其他资源发生泄漏：

```
TimeKeeper *ptk = getTimeKeeper();    // 从 TimeKeeper 层取得
                                      // 一个动态分配的对象

...                                   // 使用这个对象

delete ptk;                           // 释放它，以防资源泄漏
```

把释放工作推卸给客户将会带来出错的隐患，条目 13 中解释了这一点。关于如何修改工厂函数的接口从而防止一般的客户端错误发生，请参见条目 18。但是这些议题在此都不是主要的，这一条目中我们主要讨论的是上文中的代码存在着的一个更为基本的弱点：即使客户把每一件事都做得很完美，我们仍无法预知程序会产生怎样的行为。

现在的问题是：getTimeKeeper 返回一个指向某个派生类对象的指针（比如说 AtomicClock），这个对象最终会通过一个基类指针得到删除（比如说 TimeKeeper*指针），而基类（TimeKeeper）有一个**非虚析构函数**。这里埋藏着灾难，这是因为 C++有明确的规则：如果希望通过一个指向基类的指针来删除

一个派生类对象，并且这一基类有一个非虚析构函数，结果将是未定义的。典型的后果就是，在运行时，派生类中新派生出的部分得不到销毁。如果 getTimeKeeper 返回了一个指向 AtomicClock 对象的指针，那么这一对象中 AtomicClock 的部分（也就是 AtomicClock 类中新声明的数据成员）有可能不会被销毁掉，AtomicClock 的析构函数也可能不会得到运行。然而，这一对象中基类那一部分（也就是 TimeKeeper 的部分）很自然的会被销毁掉，这样便会产生一个古怪的“部分销毁”的对象。用这种方法来泄漏资源、破坏数据结构、浪费调试时间，实在是“再好不过”了。

排除这一问题的方法很简单：为基类提供一个虚拟的析构函数。这时删除一个派生类对象，程序就会精确地按你的需要运行了。整个对象都会得到销毁，包括所有新派生的部分：

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
  ...
};

TimeKeeper *ptk = getTimeKeeper();
...
delete ptk;                          // 现在，程序正常运转
```

通常情况下，像 TimeKeeper 这样的基类会包含除析构函数以外的虚函数，这是因为虚函数的目的是允许派生类实现中对它们进行自定义（参见条目 34）。比如说，TimeKeeper 类中可能存在一个虚函数 getCurrentTime，它在不同的派生类中将有不同的实现方式。任何有虚函数的类几乎一定都要包含一个虚析构函数。

如果一个类**不**包含虚函数，通常情况下意味着它将不作为基类使用。当一个类不作为基类时，将它的析构函数其声明为虚拟的通常情况下不是个好主意。请看下面的示例，这个类代表二维空间中的点：

```
class Point {                        // 2D 的点
public:
  Point(int xCoord, int yCoord);
```

```
  ~Point();

private:
  int x, y;
};
```

在一般情况下，如果一个 int 占用 32 比特，一个 Point 对象便可以置入一个 64 位的寄存器中。而且，这样的一个 Point 对象可以以一个 64 位数值的形式传给其他语言编写的函数，比如 C 或者 FORTRAN。然而如果 Point 的析构函数是虚拟的，那么就是另一种情况了。

虚函数的实现需要它所在的对象包含额外的信息，这一信息用来在运行时确定本对象需要调用哪个虚函数。通常，这一信息采取一个指针的形式，这个指针被称为“vptr”（“虚函数表指针”，virtual table pointer）。vptr 指向一个包含函数指针的数组，这一数组称为“vtbl”（“虚函数表”，virtual table），每个包含虚函数的类都有一个与之相关的 vtbl。当一个虚函数被一个对象调用时，就用到了该对象的 vptr 所指向的 vtbl，在 vtbl 中查找一个合适的函数指针，然后调用相应的实函数。

虚函数实现的细节并不重要。重要的**仅仅**是，如果 Point 类包含一个虚函数，这一类型的对象将会变大。在一个 32 位的架构中，Point 对象将会由 64 位（两个 int 大小）增长至 96 位（两个 int 加一个 vptr）；在 64 位架构中，Point 对象将由 64 位增长至 128 位。这是因为指向 64 位架构的指针有 64 位大小。可以看到，为 Point 添加一个 vptr 将会使对象增大 50-100%！这样，一个 64 位的寄存器便容不下一个 Point 对象了。而且，此时 C++版本的 Point 对象便不再与其它语言（比如 C 语言）有同样的结构，这是因为其它语言很可能没有 vptr 的概念。于是，除非你显式增补一个 vptr 的等价物（但这是这种语言的实现细节，而且不具备可移植性），否则 Point 对象便无法与其它语言编写的函数互通。

不得不承认，无故将所有的析构函数声明为虚拟的，与从不将它们声明为虚函数一样糟糕，这一点至关重要。实际上，许多人总结出一条解决途径：当且仅当类中至少包含一个虚函数时，要声明一个虚析构函数。

甚至在完全没有虚函数的类里，你也可能会被非虚拟的构造函数所纠缠。比如说，虽然标准的 string 类型不包含虚函数，但是误入歧途的程序员有些时候还是会将其作为基类：

```
class SpecialString: public std::string {
                                    // 这不是个好主意！
                                    // std::string 有一个非虚拟的析构函数
  ...
};
```

乍一看，这样的代码似乎没什么问题，但是如果在某应用程序里，你不知出于什么原因希望将一个 SpecialString 指针转型为 string 指针，同时你又对这个 string 指针使用了 delete，你的程序会立刻陷入未定义行为：

```
SpecialString *pss = new SpecialString("Impending Doom");

std::string *ps;
...

ps = pss;                           // SpecialString* ⇒ std::string*
...

delete ps;                          // 未定义行为！在实践中*ps 的 SpecialString
                                    // 部分资源将会泄漏，这是因为 SpecialString
                                    // 的析构函数没有被调用。
```

对于任意没有虚析构函数的类而言，上面的分析都成立，包括所有的 STL 容器类型（比如 vector、list、set、tr1::unordered_map（参见条目 54），等等）。如果你曾经继承过一个标准容器或者其他任何包含非虚析构函数的类，一定要打消这种想法！（遗憾的是，C++没有提供类似 Java 中的 final 类或 C#中的 sealed 类那种防止继承的机制）

在个别情况下，为一个类提供一个纯虚析构函数是十分方便的。你可以回忆一下，纯虚函数会使其所在的类成为抽象类——这种类不可以实例化（也就是说，你无法创建这种类型的对象）。然而某些时刻，你希望一个类成为一个抽象类，但是你有没有任何纯虚函数，这时候要怎么办呢？因为抽象类应该作为基类来使用，而基类

应该有虚析构函数，又因为纯虚函数可以造就一个抽象类，那么解决方案就显而易见了：如果你希望一个类成为一个抽象类，那么在其中声明一个纯虚析构函数。下边是示例：

```
class AWOV {                          // AWOV = "Abstract w/o Virtuals"
public:
  virtual ~AWOV() = 0;                // 声明纯虚析构函数
};
```

这个类有一个纯虚函数，所以它是一个抽象类，同时它拥有一个虚析构函数，所以你不需要担心析构函数的问题。然而这里还是有一个别扭的地方：你必须为纯虚析构函数提供一个**定义**：

```
AWOV::~AWOV() {}                      // 纯虚析构函数的定义
```

析构函数的工作方式是这样的：首先调用最后派生出的类的析构函数，然后依次调用上一层基类的析构函数。由于当调用一个 AWOV 的派生类的析构函数时，编译器会自动调用~AWOV，因此你必须为~AWOV 提供一个函数体。否则连接器将会报错。

为基类提供虚析构函数的原则仅对多态基类（这种基类允许通过其接口来操控派生类的类型）有效。我们说 TimeKeeper 是一个多态基类，这是由于即使我们手头只有 TimeKeeper 指向它们的指针，我们仍可以对 AtomicClock 和 WaterClock 进行操控。

并不是所有的基类都要具有多态性。比如说，标准 string 类型、STL 容器都不用作基类，因此它们都不具备多态性。另外有一些类是设计用作基类的，但是它们并未被设计成多态类。这些类（例如条目 6 中的 Uncopyable 和标准类中的 input_iterator_tag（参见条目 47））不允许通过其接口来操控它的派生类。因此，它们并不需要虚析构函数。

时刻牢记

- 应该为多态基类声明虚析构函数。一旦一个类包含虚函数，它就应该包含一个虚析构函数。
- 如果一个类不用作基类或者不需具有多态性，便不应该为它声明虚析构函数。

条目8： 防止因异常中止析构函数

C++并没有禁止析构函数引发异常，但是 C++无疑不会推荐这一做法。这样做有充足的理由。请看下边的代码：

```
class Widget {
public:
  ...
  ~Widget() { ... }                    // 假设它会引发一个异常
};

void doSomething()
{
  std::vector<Widget> v;
  ...
}                                      // v 在这里被自动销毁
```

当 vector v 被销毁时，它也有责任销毁其所包含的所有的 Widget。假设 v 中包含十个 Widget，并且在对第一个进行析构时抛出了一个异常。那么剩下的九个 Widget 则仍需要得到销毁（否则它们所占有的资源就会发生泄漏），所以 v 应该为所有剩下的 Widget——调用析构函数。但是假设在对这些对象进行销毁时，又出现了第二个 Widget 抛出了一个异常，现在同时存在着两个活动的异常，这对于 C++来说已经是太多了。在极端巧合的情形下，程序中同时出现了两个活动的异常，此时程序的运行要么会中止，要么会产生未定义行为。本示例将产生未定义行为。在使用其它的标准库容器（比如 list、set 等），任意的 TR1 容器（参见条目 54），甚至是一个数组，同样都会产生未定义行为。然而为你带来麻烦的不仅仅是这些容器或者数组，析构函数抛出异常会引发不成熟的程序终止或者未定义行为，甚至在没有容器和数组的情况下也会发生。C++**不**喜欢能够引发异常的析构函数！

这个问题很好理解，但是当你的析构函数的某一操作可能失败，并且有可能抛出一个异常时，你应该怎么做呢？请看下边的示例，其中假设你使用一个类进行数据库连接：

```
class DBConnection {
public:
```

```
  ...
  static DBConnection create();     // 返回 DBConnection 对象的函数；
                                    // 为简化代码省略了参数表

  void close();                     // 关闭连接；若关闭失败则抛出异常
};
```

为了确保客户不会忘记为 DBConnection 对象调用 close 函数，一个可行的方案是：创建一个新的类来管理 DBConnection 的资源，在这个类的析构函数中调用 close。这种资源管理类在第三章中作详细的介绍，在本节中，我们仅关心这些类的析构函数是什么样的：

```
class DBConn {                      // 该类用来管理 DBConnection 对象的资源
public:
  ...
  ~DBConn()                         // 确保数据库连接总能关闭
  {
    db.close();
  }
private:
  DBConnection db;
};
```

客户可以这样编写：

```
{                                   // 开始一个程序块
   DBConn dbc(DBConnection::create());
                                    // 创建一个 DBConnection 对象，然后
                                    // 把它交给一个 DBConn 对象来管理

...                                 // 通过 DBConn 的接口使用这个
                                    // DBConnection 对象

}                                   // 在该程序块的最后，这个 DBConn 对象
                                    // 被销毁了，就好像自动调用了那个
                                    // DBConnection 对象的 close 函数
```

只要对 close 的调用能成功，这个 DBConn 的方案就是一个好主意，但是一旦这一调用会引发一个异常， DBConn 的析构函数则会使这个异常蔓延开来，也就是所谓的，允许“因异常而中止析构函数”。这便是问题所在，因为析构函数在此处抛出异常意味着麻烦将会出现。

避免这类麻烦有两种主要的办法。DBConn 的析构函数可以：

- **终止程序**——如果 close 抛出异常则终止程序，通常通过调用 abort 实现：

```
DBConn::~DBConn()
{
  try { db.close(); }
  catch (...) {
    在日志上记载：调用 close 失败;
    std::abort();
  }
}
```

  如果在析构过程中发生了一个错误，从而程序无法继续运行，上面的方法就是一个可行的选择。如果允许析构函数传播异常将导致程序产生未定义行为，这样做的优势就在于可以避免类似的事情发生。也就是说，调用 abort 函数可以防止程序产生未定义行为。

- **吞掉这个异常**——由调用 close 函数产生的异常

```
DBConn::~DBConn()
{
  try { db.close(); }
  catch (...) {
      在日志上记载：调用 close 失败;
  }
}
```

  在大多数情况下，忽略掉异常的存在并不是一个好主意，因为这样做你会错过一些重要的信息——**一些东西出错了！**然而在某些时刻，忽略异常比让程序担上不成熟终止或未定义行为的风险要强一些。为了让忽略异常成为一个可行的方案，程序必须有能力在忽略刚发生的错误之后仍然可以稳定地继续运行。

这两个方案都不是那么的动人心弦。它们存在着同样的问题，即二者都没有办法在第一时间对 close 抛出的异常做出反应。

一个更好的策略是：改进 DBConn 接口的设计，使得客户有机会自己处理可能发生的问题。举例说，DBConn 可以自己提供一个 close 函数，这样就为客户提供了途径来处理由 DBConnection 的 close 产生的异常。这样做还可以跟踪 DBConnection 所建立的连接是否被 DBConnection 自己的 close 函数正常关闭，如果关闭失败则在 DBConn 的析构函数将其关闭。这可以防止已建立的连接发生泄漏。然而，如果在 DBConn 的析构函数中对 close 的调用仍然不成功，我们还是需要中止运行或者忽略异常。

```
class DBConn {
public:
  ...
  void close()                          // 新函数，供客户调用
  {
    db.close();
    closed = true;
  }

  ~DBConn() {
     if (!closed) {
       try {
         db.close();                    // 如果客户没有关闭连接，则在这里关闭它
       }
       catch (...) {                    // 如果没有正常关闭，
         在日志上记载：调用 close 失败;  // 首先做好记录，
         ...                            // 然后终止或吞掉该异常
      }
    }
  }

private:
  DBConnection db;
  bool closed;
```

```
};
```

调用 close 的责任原本是 DBConn 的析构函数的，而现在我们却将其转交给 DBConn 的客户（DBConn 的析构函数还包含一个“备用的”调用）。可能你会认为这样做实属毫无顾忌地推卸责任，你甚至可能认为这是对“让接口更易于正确用”这一忠告（见条目 18）的违背。实际上，两者都不是。如果一个操作可能由于一次异常的抛出而失败，同时这个异常有必要得到处理，这一异常**不应该来自析构函数**。这是因为引发异常的析构函数是十分危险的，它使你的程序始终位于风口浪尖：你无法避免不成熟的终止和未定义行为的风险。在上边的示例中，让客户自己手动调用 close 并不会为其带来过多的负担，相反地，这样作为客户提供了处理那些原本他们无法接触的错误的机会。如果他们没有发现这个机会的裨益所在（可能是因为他们相信错误不会发生得这么巧），他们可以忽略这个机会，然后依赖 DBConn 的析构函数为他们调用 close 函数。如果就在这一刻发生了错误（也就是说 close 确实抛出了异常）DBConn 会忽略这个异常或者终止程序。客户对此也没有什么好抱怨的，毕竟，在处理问题时是他们犯下了第一个错误，是他们自己选择不去利用它的。

时刻牢记

- 永远不要让析构函数引发异常。如果析构函数所调用的函数会抛出异常的话，那么析构函数中要捕捉到所有异常，然后忽略它们或者终止程序。
- 在一次操作中，如果类的客户有必要对抛出的异常做出反应，那么这个类应该提供一个常规的函数（而不是析构函数）来进行这一操作。

条目9： 永远不要在构造或析构的过程中调用虚函数

让我们直切正题：在程序进行构造或析构期间，你绝不能调用虚函数，这是因为这样的调用并不会按你所期望的执行，即使能够顺利执行，你也不会觉得十分舒服。如果你曾经是一个 Java 或 C#的程序员，并且在最近期望返回 C++的怀抱，那么请你格外留意本条目，因为在这一问题上，C++与其他语言走的是完全不同的两条路线。

假设有一个股票交易模拟系统，你为它编写了一个类的层次化结构，其中包括实现购买、抛售等功能的类。这类交易应该是可以审计的，这一点很重要，所以说每创建一次交易时，都应该在日志中创建一条审计相关内容的记录。下面是一个看似合理的解决方案：

```
class Transaction {                          // 所有交易的基类
public:
  Transaction();
  virtual void logTransaction() const = 0;   // 作类型相关的记录
  ...
};

Transaction::Transaction()                   // 基类构造函数的实现
{
  ...
  logTransaction();                          // 最后，记录这次交易
}

class BuyTransaction: public Transaction {   // 派生类
public:
  virtual void logTransaction() const;       // 当前类型交易是如何记录的
  ...
};

class SellTransaction: public Transaction {  // 派生类
public:
  virtual void logTransaction() const;       // 当前类型交易是如何记录的
  ...
};
```

请考虑一下在下边的代码运行时会发生什么：

```
BuyTransaction b;
```

很明显的是此时 BuyTransaction 的构造函数将被调用，但是，首先必须调用 Transaction 的构造函数。对于一个派生的对象，其基类那一部分会首先得到构造，然后才是派生类的部分。Transaction 的构造函数中最后一行调用了虚函数

logTransaction，意外的事情就从这里发生了：此处调用的是 Translation 版本的 logTransaction 函数，而**不是** BuyTransaction 版本的——即使此处创建的对象是 BuyTransaction 类型的。在基类部分的构造过程中，虚函数永远也不会尝试去匹配派生类部分。取而代之的是，对象仍然保持基类的行为。更随意一点的说法是，在基类部分构造的过程中，虚函数并不会被构造。

这一行为看上去匪夷所思，但是这里有很充足的理由来解释它。由于基类的构造函数先于派生类运行，在基类构造函数运行的时候，派生类的数据成员还没有被初始化。如果在基类构造函数向下匹配派生类时调用了虚函数，那么基类的函数几乎一定会调用局部数据成员，但此时这些数据成员此时尚未得到初始化。你的程序将会出现无尽的未定义行为，你也会在整夜受到琐碎的调试工作的折磨。当一个对象中某些部分尚未初始化的时候，此时对其进行调用会存在内在的危险，所以 C++不允许你这样做。

实际情况比上文介绍的更为基础。对于一个派生类的对象来说，在其进行基类部分构造工作的时候，这一对象的类型**就是**基类的。不仅仅虚函数会解析为基类的，而且 C++中"使用运行时类型信息"的部分（比如 dynamic_cast（参见条目 27）和 typeid）也会将其看作基类类型的对象。在我们的示例中，当调用 Transaction 的构造函数以初始化一个 BuyTransaction 对象的基类部分时，这一对象是 Transaction 类型的。C++的任何一部分都会这样处理，这种处理方式是有意义的：由于这个对象的 BuyTransaction 部分尚未得到初始化，所以假定它们不存在才是最安全的处理方法。对于一个派生类对象来说，只有派生类的构造函数开始执行，这个对象才会变成该派生类的对象。

对于析构过程可以应用同样的推理方式。一旦派生类的析构函数运行完毕，对象中派生类的那一部分数据成员将取得未定义的值，所以 C++会认为它们不再存在。在进入基类的析构函数时，这个对象将成为一个基类对象，C++的所有部分——包括虚函数、dynamic_cast 等等——都会这样对待该对象。

在上文的示例代码中，Transaction 的构造函数对一个虚函数进行了一次直接调用，很显然这样做是违背本条中的指导方针的。这样的违规实在太容易发现了，一些编译器都会对其做出警告。（其他一些则不会。参见条目 53 对编译器警告信息的讨论）即使没有警告，问题也一定会在运行之前变得很明显，这是因为

Transaction 中的 logTransaction 函数是纯虚函数，除非它得到了定义（不像是真的，但存在这种可能，参见条目 34），程序才有可能会得到连接，其他情况都会报错：连接器无法找到必要的 Transaction::logTransaction 的具体实现。

查找构造或析构过程中对虚函数的调用并不总是一帆风顺的。如果 Transaction 拥有多个构造函数，它们所进行的工作中有一部分是相同的，那么可以将这些公共的初始化代码（包括对 logTransaction 的调用）放入一个私有的非虚拟的初始化函数中，这样做可以避免代码重复，从软件工程角度来讲这似乎是一个很好的做法，我们将这一函数命名为 init：

```
class Transaction {
public:
  Transaction()
  { init(); }                          // 调用非虚函数...

  virtual void logTransaction() const = 0;
  ...

private:
  void init()
  {
    ...
    logTransaction();                  // ...而它却调用一个虚函数！
  }
};
```

这样的代码与前文中的版本使用的是同一理念，但是这样做所带来的危害更为隐蔽和严重，这是因为这样的代码会得到正常的编译和连接而不会报错。这种情况下，由于 logTransaction 是 Transaction 中的一个纯虚函数，大多数运行时系统将会在调用这个纯虚函数时中止程序（通常情况下会针对这一结果显示出一个消息）。然而如果 logTransaction 是一个“正常的”虚函数（也就是说，不是纯虚的），并且在 Transaction 中给出了一些实现，那么此时将调用这一版本的 logTranscation，程序将会“愉快地一路小跑”下去，至于为什么在创建派生类对象时会调用错误的 logTransaction 版本，程序可就不管这一套了。避免这类问

题的唯一途径就是：在正在创建或销毁的对象的构造函数和析构函数中，确保永远不要调用虚函数，对于构造函数和析构函数所调用的所有函数都应遵守这一约定。

那么，每当创建一个 Transaction 层次结构中的对象时，如何**确保**去调用正确的 logTransaction 版本呢？显然地，在 Transaction 的构造函数中调用一个虚函数是一个错误的做法。

为解决这一问题我们可以另辟蹊径。方案之一就是：将 Transaction 中的 logTransaction 变为一个非虚函数，然后要求派生类的构造函数把必要的日志记录传递给 Transaction 的构造函数。这个构造函数对于非虚 logTransaction 的调用就是安全的。就像这样：

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);

  void logTransaction(const std::string& logInfo) const;
                                    // 现在 logTransaction 是非虚函数
  ...
};

Transaction::Transaction(const std::string& logInfo)
{
  ...
  logTransaction(logInfo);          // 现在调用的是一个非虚函数
}

class BuyTransaction: public Transaction {
public:
 BuyTransaction( 参数表 )
 : Transaction(createLogString( 参数表 ))
  { ... }                           // 将记录传递给基类构造函数

   ...

private:
```

```
  static std::string createLogString( 参数表 );
};
```

换句话说，你不能使用虚函数在基类构造过程中向下调用派生类的部分，作为一种补偿，你可以让派生类将一些必要的构造信息向上传递给基类的构造函数。

请注意上述示例里 BuyTransaction 类中（私有的）静态函数 createLogString 的使用。这里使用了一个辅助函数创建一个值来传递给基类的构造函数，通常情况下这样做更为方便（而且更具备可读性），这样做使为基类提供所需信息的成员初始化表变得更加直观。这是因为这样做解决了“为基类提供所需信息的成员初始化表”不直观的问题。意外调用初生的 BuyTransaction 对象中那些尚未初始化的数据成员是十分危险的，由于 createLogString 是静态的，此处便不存在这一危险。这一点很重要，因为这些数据成员正处于未定义的状态，这一事实便解释了为什么“在基类部分构造或析构期间调用虚函数，不会在第一时间向下匹配派生类”。

时刻牢记

- 不要在构造和析构的过程中调用虚函数，因为这样的调用永远不会转向当前执行的析构函数或构造函数更深层的派生类中执行。

条目10： 让赋值运算符返回一个指向*this 的引用

关于赋值有许多有趣的事情，其中之一就是：你可以把赋值操作连在一起：

```
int x, y, z;
x = y = z = 15;                          // 一连串的赋值
```

另一件有趣的事是：这一赋值工作是自右结合的，所以上面的赋值链可以解析成这样：

```
x = (y = (z = 15));
```

在这里，15 首先赋值给 z，然后这次赋值的结果（就是更新过的 z 的值）将赋给 y，然后这次赋值的结果（就是更新过的 y 的值）又赋给 x。

这种实现方法的本质是：赋值时，返回一个指向运算符左手边参数的引用，当你为你的类实现赋值运算符时，你应该遵循这一惯例：

```
class Widget {
public:
  ...
Widget& operator=(const Widget& rhs)   // 返回值类型是一个指向当前类的引用，
{
  ...
  return *this;                        // 返回运算符左边的对象
  }
  ...
};
```

这一惯例对于所有的赋值运算符同样适用，不仅仅是上述的标准形式。于是，我们要这样书写：

```
class Widget {
public:
  ...
  Widget& operator+=(const Widget& rhs)
  {                                    // 这一惯例对于+=、-=、*=等运算符均适用
     ...
   return *this;
  }
   Widget& operator=(int rhs)          // 即使某些时刻运算符的参数不符合惯例，
   {                                   // 赋值运算符仍应遵守这一惯例
      ...
      return *this;
   }
   ...
};
```

这仅仅是一个惯例，不遵循这一惯例的代码也能够得到编译。然而，所有的内建数据类型，以及标准库中所包括（或者即将包括的，可以参见条目 54）的所有类型（比如说，string、vector、complex、tr1::shared_ptr，等等）都遵守这

一惯例。当你要做一些有悖于该惯例的事情时先要考虑一下，是否有充分的理由这样做？否则请遵循惯例。

时刻牢记

- 让赋值运算符返回一个指向`*this`的引用。

条目11： 在operator=中要处理自赋值问题

当对象为其自身赋值时，就发生了一次“自赋值”：

```
class Widget { ... };

Widget w;
...
w = w;                                  // 自赋值
```

这样做看上去很愚蠢，但是却是合法的，因此客户一定会这样操做。而且，自赋值本身并不总是那么容易辨认的。比如：

```
a[i] = a[j];                            // 潜在的自赋值
```

如果`i`和`j`的值相同，那么这就是一次自赋值。另外

```
*px = *py;                              // 潜在的自赋值
```

在`px`和`py`指向同一处时，上面的代码也是一次自赋值。这些自赋值并不是那么一目了然，它们是由**别名**造成的：可以通过多种方式引用同一个对象。一般来说，用来操作指向同一类型多个对象的引用或指针的代码都应考虑对象重复的问题。实际上，假如两个对象来自同一层次，即使它们并未声明为同一类型，也要考虑重复问题，这是因为一个基类的引用或指针可以引用或指向其派生类的类型的对象。

```
class Base { ... };

class Derived: public Base { ... };

void doSomething(const Base& rb,  // rb 和 *pd 可能实际上
```

```
                    Derived* pd);      // 是同一个对象
```

假设你遵循条目 13 和条目 14 中的建议，你将会一直使用对象来管理资源，而且在复制时你将会确保资源管理对象能正确工作。如果上边的假设成立，你的赋值运算符很可能在处理自赋值时将是安全的，你不需要额外关注它。然而如果你试图自己来管理资源（你在编写资源管理类时一定会这样做），此时你很有可能落入这个陷阱中：一个对象尚未用完，但是你却不小心将其释放了。比如说，你创建了一个类，其中放置了一个原始指针来动态分配位图：

```
class Bitmap { ... };

class Widget {
  ...

private:
  Bitmap *pb;                          // 指向一个分配在堆上的对象
};
```

下边给出 operator=的一个实现，它在表面看上去很合理，但是如果存在自赋值，它便是不安全的。（它也不是异常安全的，稍后我们讨论这个问题）

```
Widget&
Widget::operator=(const Widget& rhs)    // operator= 不安全的实现
{
  delete pb;                             // 停止使用当前的位图
  pb = new Bitmap(*rhs.pb);              // 开始使用 rhs 位图的一份拷贝
  return *this;                          // 参见条目 10
}
```

此处的自赋值问题出现在 operator=的内部，*this（赋值操作的目标）和 rhs 有可能是同一对象。如果它们是，delete 便不仅仅销毁了当前对象的位图，同时它也销毁了 rhs 的位图。Widget 的值本不应该在自赋值操作中改变，然而在函数的末尾，它会发现其包含的指针指向了一个已经被删除的对象！

防止这类错误发生的传统方法是：在 operator=的最顶端通过**一致性检测**来监视自赋值：

```
Widget& Widget::operator=(const Widget& rhs)
{
  if (this == &rhs) return *this;  // 一致性检测： 如果出现自赋值则什么也不做
  delete pb;
  pb = new Bitmap(*rhs.pb);
  return *this;
}
```

这样可以正常工作，但是上文提到的 operator=先前的版本不仅仅是自赋值不安全的，同时也是异常不安全的。尤其在“new Bitmap”语句引发了一个异常（有可能是可分配内存耗尽，或者是 Bitmap 的拷贝构造函数抛出了一个异常）时，Widget 最终将包含一个指向已被删除的 Bitmap 的指针。这类指针是有毒的。你无法安全的删除它们。你甚至没办法安全的读取它们。此时你所做的唯一一件安全的事情也许就是耗费大量精力调试程序来寻找这些指针的出处。

还好，在让 operator=在做到异常安全的同时，它同时也能做到自赋值安全。因此，忽略自赋值的问题而把目光集中在异常安全的问题上，就愈加合理了。条目 29 中深入讨论异常安全的问题，但是通过本条已足以看出：在许多情况下，认真安排语句的次序可以使你的代码做到异常安全（同时也是自赋值安全的）。比方说，这里我们只需要认真考虑：在我们没有把 pb 所指向的对象复制出来以前，千万不要删除它：

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb;                 // 记忆原始的 pb
  pb = new Bitmap(*rhs.pb);           // 让 pb 指向*pb 的一个副本
  delete pOrig;                       // 删除原始的 pb

  return *this;
}
```

现在，如果“new Bitmap”抛出一个异常，pb（及其所在的 Widget）就不会被改动。即使没有进行一致性检测，这段代码也可以解决自赋值问题，这是因为我们为原始位图复制出了一个副本，并将原始版本删除，然后让 pb 指向我们复制出的那个副本。这也许不是解决自赋值问题的最高效的途径，但是这样做确实有效。

如果你考虑到效率问题，你可以重新考虑在程序开端添加一致性检测。然而在这样做之前先问一下自己：你预计自赋值出现的有多频繁，因为一致性检测也有系统开销。首先它使得代码（源代码和对象）的体积变得稍大一些，其次它也会为控制流引入一个分支，二者都会降低运行的速度。比如说，指令预读、捕获、管线分配等操作的执行效率将会受到影响。

在 operator=中手动安排语句可以确保实现同时做到异常安全和自赋值安全，这里有一个替代方案：使用一个称为"复制并 swap"的技术。由于这一技术更加贴近异常安全的议题，因此我们在条目 29 中讨论它。这是编写 operator=的一个相当普遍的方法，这里看一下它一般实现的方法是值得的：

```
class Widget {
  ...
  void swap(Widget& rhs);          // 交换*this 和 rhs 中的数据;
  ...                              // 更多细节请参见条目 29
};

Widget& Widget::operator=(const Widget& rhs)
{
  Widget temp(rhs);                // 为 rhs 的数据保存副本
  swap(temp);                      // 使用上边的副本与*this 交换
  return *this;
}
```

通过利用这些事实：(1)一个类的拷贝赋值运算符可以通过传值方式传参；(2)通过传值即可传递某参数的**副本**（参见条目 20），上述主题可以进行以下演化：

```
Widget& Widget::operator=(Widget rhs)  // rhs 是传进对象的一份拷贝
{                                      // 请注意此处是传值方式传递参数

  swap(rhs);                           // 交换*this 与 rhs 的数据
  return *this;
}
```

从个人角度来讲，我很担心这一做法会通过牺牲清晰度来换取灵巧性，但是把复制操作从函数体中移出来，放在参数构造的过程中，在一些场合确实能够让编译器生成更加高效的代码。

时刻牢记

- 在一个对象为自己赋值时，要确保 operator=可以正常地运行。可以使用的技术有 :比较源对象和目标对象的地址、谨慎安排语句顺序、以及“复制并 swap”。
- 在两个或两个以上的对象完全一样时，要确保对于这些重复对象的操作可以正常运行。

条目12： 要复制整个对象，不要遗漏任一部分

在一个设计良好的面向对象系统中，对象的所有内在部分都会被封装起来，只有两个函数是用来复制对象的 :即拷贝构造函数和拷贝赋值运算符，这两个函数的功能恰如其名，我们将它们称为**拷贝函数**。条目 5 中详细讲述了编译器将会在必要时自动生成拷贝函数，然后该条目还说明了编译器生成的版本可以精确的按你所预期的执行：当前正在复制的对象的所有数据都会得到复制。

当你声明你自己的拷贝函数时，你就向编译器表明，默认实现方式中有一些内容你不喜欢。编译器会把这种做法视为对它的冒犯，它会以一个古怪的方式来报复你：当你的实现几乎一定要出现错误时，它就是不告诉你。

下面示例中是一个表示顾客的类，其中拷贝函数是手动编写的，以便将对它们的调用记入日志：

```
void logCall(const std::string& funcName);
                                    // 创建一个日志记录
class Customer {
public:
  ...
  Customer(const Customer& rhs);
  Customer& operator=(const Customer& rhs);
  ...
```

```
private:
  std::string name;
};

Customer::Customer(const Customer& rhs)
: name(rhs.name)                    // 复制 rhs 中的数据
{
  logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
  logCall("Customer copy assignment operator");
  name = rhs.name;                  // 复制 rhs 中的数据
  return *this;                     // 参见条目 10
}
```

这里看上去一切正常，而且实际上一切确实是正常的——但当另一个数据成员添加入 Customer 时，意外就发生了：

```
class Date { ... };                 // 记录日期

class Customer {
public:
  ...                               // 同上

private:
  std::string name;
  Date lastTransaction;
};
```

现在，前面的拷贝函数将进行**部分复制**：它们会复制出顾客的姓名（name），但是不会复制最后一次交易（lastTransaction）。迄今为止大多数编译器对此视而不见，即使将警告调至最大模式也不会报出任何信息（另请参见条目 53）。如果你自己编写拷贝函数，这便是这些编译器对你的“复仇”。由于你拒绝了编译器提供的

拷贝函数，那么编译器就拒绝在你的代码不完整时通知你。结果很明显：如果你为一个类添加了一个数据成员，你必须确保更新相应的拷贝函数。（同时你也需要更新所有的构造函数（参见条目 4 和条目 45），以及类中所有的非标准格式的 operator=(参见条目 10 中的示例)。如果你忘记了，编译器也不会及时提醒你。）

通过继承，这一问题可以带来更加严重却隐蔽的危害，请考虑下边的示例：

```
class PriorityCustomer: public Customer {  // 一个派生类
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...

private:
   int priority;
};

PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");
  priority = rhs.priority;
  return *this;
}
```

PriorityCustomer 的拷贝函数看上去能复制 PriorityCustomer 中的所有数据，但是请仔细看一下，的确，这些拷贝函数确实能够复制 PriorityCustomer 中声明的数据成员，但是每一个 PriorityCustomer 对象都还包含继承自 Customer 的所有数据成员，这些数据成员始终没有得到复制！

PriorityCustomer 的拷贝构造函数并没有指明任何参数去传递至基类的构造函数（也就是说，它在成员初始化表中从未提及 Customer），于是 PriorityCustomer 中 Customer 那一部分将由 Customer 的无参构造函数——默认构造函数（假设存在一个，如果没有编译器将报错）进行初始化。这一构造函数将为 name 和 lastTransation 进行**默认**的初始化。

对于 PriorityCustomer 的拷贝赋值运算符而言，情况有小小的不同。在任何情况下它都不会尝试去修改其基类的数据成员，所以这些数据成员不会得到更新。

一旦你亲自为一个继承类编写了拷贝函数，你必须同时留心其基类的部分。当然这些部分通常情况下是私有的，所以你无法直接访问它们。取而代之的是，派生类的拷贝函数必须调用这些私有数据在基类中相关的函数：

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: Customer(rhs),                    // 调用基类的拷贝构造函数
  priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");
  Customer::operator=(rhs);         // 为基类部分赋值
  priority = rhs.priority;
  return *this;
}
```

本条目标题中的“不要遗漏任何部分”在这里就显得很清晰了。当你编写拷贝函数时，要确认：(1)复制所有的局部数据成员，(2) 对所有的基类调用适当的拷贝函数。

从实践角度讲，这两个拷贝函数通常会很相似，这似乎会怂恿你去尝试让一个函数去调用另一个来避免代码重复。你避免代码重复的渴望之心是值得称赞的，但是让一个拷贝函数调用另一个并不是一个好的实现方式。

让拷贝赋值运算符调用拷贝构造函数是没有任何意义的，这是因为你是在尝试构造一个已经存在的对象。这实在是毫无意义，甚至没有语法支持这样做。即使存在语法**看上去**可以让你这样做，但事实上你达不到预期的目的；同时存在语法确实可以迂回实现，但是却会在一些情况下破坏你的对象。所以我不会向你介绍这些语法。你只需清楚让拷贝赋值运算符去调用拷贝构造函数不是一个好主意就可以了。

让我们逆向考虑此问题——让拷贝构造函数去调用拷贝赋值运算符——这样做同样是毫无意义的。一个构造函数初始化新的对象，但是一个赋值运算符仅仅可以应用于已经初始化的对象。对于一个正在构造中的对象而言，对其进行赋值操作就意味着对未初始化的对象进行操作（但是这些操作仅对已初始化的对象起作用）。这是很荒唐的，请不要尝试。

如果你发现你的拷贝构造函数和拷贝赋值运算符很相似时，如果你希望排除重复代码，可以通过创建第三个成员函数供两者调用来取代上面的方法。通常情况下，这样的函数应该是私有的，一般将其命名为 `init`。这样的策略是安全的，排除拷贝构造函数和拷贝赋值运算符中的代码重复，这是一个经过证实安全有效的方法。

时刻牢记

- 要确保拷贝函数拷贝对象的所有的数据成员，及其基类的所有部分，不要有遗漏。
- 不要尝试去实现一个拷贝函数来供其它的拷贝函数调用。你应该把公共部分放入一个“第三方函数”中共所有拷贝函数调用。

第三章. 资源管理

资源是这样一种东西：一旦你借助它们所做的事情完成了，你必须要将其返回给系统。如果你没有这样做，那么不好的事情就会发生。在 C++程序中，最常用的资源是动态分配的内存（如果你分配了内存但是却忘记了释放它们，你的程序就会遇到一次内存泄漏），但是内存只是你所需要管理的众多资源中的一种。其它常见的资源包括文件描述符、互斥锁、图形用户界面（GUI）中的字体和画笔、数据库联接、以及网络套结字等等。无论是何种资源，在你借助它所做的工作完成以后都要将其释放，这一点是很重要的。

试图手动将资源管理得井井有条，在任何情况下都是很困难的事情。但当问题转向异常处理、多路返回函数、以及维护程序员在未对其所作的修改有充分理解之前对程序作出的，你就会清楚地发现，专门用来解决资源管理问题的方法并不是很充足。

本章以介绍一个基于对象的资源管理方法开始，该方法构建于 C++所支持的构造函数、析构函数、拷贝操作符之上。实践显示，如果严格坚持这一方法，便可以解决几乎所有的资源管理问题。本章靠后的条目将专门讲解内存管理的问题。这些条目是对前边较为一般化的条目的补充，因为管理内存的对象需要搞清楚如何恰当的工作。

条目13： 要使用对象来管理资源

在下面的示例中，我们的工作将围绕一个为投资（或者是股票、证券等等）建模的库展开，在这个库中，各种各样的投资类型都继承自同一个基类——Investment：

```
class Investment { ... };           // 投资类型层次结构的基类
```

继续上面的示例，我们考虑通过工厂函数(参见条目 7)来提供具体的 Investment 对象：

```
Investment* createInvestment();    // 返回一个指针，指向 Investment
                                   // 层次结构中动态分配的对象，
                                   // 调用者必须要自行将其删除
                                   //  （省略参数表以简化代码）
```

从上面代码中的注释中可以看出，当调用者完成对于 createInvestment 函数返回对象的操作后，有义务删除这一对象。请看下边代码中一个履行这一义务的函数——f：

```
void f()
{
  Investment *pInv = createInvestment(); // 调用工厂函数
  ...                                    // 使用 pInv
  delete pInv;                           // 释放该对象
}
```

这看上去可以正常运行，但是在一些情况下 f 可能无法成功删除它从 createInvestment 获得的对象。在上述代码的“...”部分中的某处可能存在不成熟的 return 语句。一旦这样的 return 语句得到了执行，那么程序将永远无法到达 delete 语句。当 createInvestment 和 delete 在一个循环内使用时，也会出现类似的情形，这样的循环有可能在遇到 continue 或 goto 语句而提前退出。最后，“...”中的一些语句还有可能抛出异常。如果真的有异常抛出，程序同样也不会达到 delete。无论 delete 是如何被跳过的，包含 Investment 对象的内存都将泄漏，同时这些对象所包含的资源都有可能得不到释放。

当然，用心编程就有可能防止这类错误发生，但是请想象一下代码会多么的不固定——你需要不停地修改代码。随着软件不断得到维护，为一个函数添加 return 或 continue 语句可能会对其资源管理策略造成怎样的影响呢，一些人可能由于不完全理解这一问题就这样做了。还有更严重的，就是 f 函数的“...”部分可能调用了一个这样的函数：它原先不会抛出异常，但在其得到“改进”之后，它突然又开始能够抛出异常了。寄希望于 f 函数总能达到其中的 delete 语句并不可靠。

为了确保 createInvestment 所返回的资源总能得到释放，我们需要将这类资源放置在一个对象中，并让该对象的析构函数在程序离开 f 时自动释放资源。实际上，这就是本条目蕴含理念的一半了：通过将资源放置在对象中，我们可以依赖 C++对默认析构函数的自动调用来确保资源及时得到释放。(另一半理念稍后讲解。)

许多资源是在堆上动态分配的，并且仅仅在单一的程序块或函数中使用，这类资源应该在程序离开这一程序块或函数之前得到释放。标准库中的 auto_ptr 就是为这类情况量身定做的。auto_ptr 是一个类似于指针的对象（**智能指针**），其析构

函数可以自动对其所指内容执行 delete。以下代码描述了如何使用 auto_ptr 来防止 f 潜在的资源泄漏。

```
void f()
{
  std::auto_ptr<Investment> pInv(createInvestment());
                                    // 调用工厂函数

  ...                               // 像前文示例一样使用 pInv

}                                   // 通过 auto_ptr 的析构函数
                                    // 自动删除 pInv
```

这一简单的示例向我们展示了使用对象管理资源的两大关键问题：

- **获取资源后，资源将立即转交给资源管理对象。**上边的示例中，createInvestment 返回的资源将初始化一个 auto_ptr，从而实现对这类资源的管理。事实上，使用对象来管理资源的理念通常称为“资源获取即初始化”（Resource Acquisition Is Initialization，简称 RAII），这是因为，在同一个语句中获取一个资源并且初始化一个资源管理对象是很平常的。某些时候获取的资源就是会赋值给一个资源管理对象，而不是初始化。但是无论是哪种途径，在获取到一个资源时，每个资源都都会立即转向一个资源管理对象。

- **资源管理对象使用其析构函数来确保资源得到释放。**由于析构函数是在对象销毁时自动调用的（比如，当对象在其作用域外时），所以不管程序是如何离开一个块的，资源都会被正确地释放。如果释放资源会带来异常，那么事情就会变得错综复杂。但那是条目 8 中介绍的内容，我们这里不关心这些。

由于当一个 auto_ptr 被销毁时，它将自动删除其所指向的内容，所以永远不存在多个 auto_ptr 指向同一个对象的情况，这一点很重要。如果存在的话，这个对象就会被多次删除，这样你的程序就会立即陷入未定义行为。为了防止此类问题发生，auto_ptr 有一个不同寻常的特性：如果你复制它们（通过拷贝构造函数或者拷贝赋值运算符），它们就会被重设为 null，然后资源的所有权将由复制出的指针独占！

```
std::auto_ptr<Investment> pInv1(createInvestment());
```

```
                                    // pInv1 指向 createInvestment
                                    // 所返回的对象

std::auto_ptr<Investment> pInv2(pInv1);
                                    // 现在 pInv2 指向这一对象，
                                    // pInv1 被重设为 null

pInv1 = pInv2;                      // 现在 pInv1 指向这一对象
                                    // pInv2 被重设为 null
```

在这一古怪的复制方法中，由于 auto_ptr 必须仅仅指向一个资源，因此增加了对于资源管理的潜在需求。这意味着 auto_ptr 并不适合于所有动态分配的资源。比如说，STL 容器要求其内容表现出“正常”的复制行为，所以在容器中放置 auto_ptr 是不允许的。

引用计数智能指针（*reference-counting smart pointer*，简称 RCSP）是 auto_ptr 的一个替代品。RCSP 是这样的智能指针：它可以跟踪有多少的对象指向了一个特定的资源，同时当没有指针在指向这一资源时，智能指针会自动删除该资源。可以看出，RCSP 的行为与垃圾回收器很相似。然而，与垃圾回收器不同的是，RCSP 不能够打断循环引用（比如两个不同的、空闲的、互相指向对方的对象）。

TR1 的 tr1::shared_ptr 就是一个 RCSP，于是你可以按下面的方式来编写 f：

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>
    pInv(createInvestment());       // 调用工厂函数

  ...                               // 像前文示例一样使用 pInv

}                                   // 通过 shared_ptr 的析构函数
                                    // 自动删除 pInv
```

上面的代码与使用 auto_ptr 时几乎完全相同，但是复制 shared_ptr 的行为更加自然：

```
void f()
{
  ...

  std::tr1::shared_ptr<Investment> pInv1(createInvestment());
                                   // pInv1 指向 createInvestment
                                   // 所返回的对象

  std::tr1::shared_ptr<Investment> pInv2(pInv1);
                                   // 现在 pInv1 与 pInv2 均指向同一对象

  pInv1 = pInv2;                   // 同上 — 因为什么都没有改变
  ...
}                                  // pInv1 与 pInv2 被销毁，
                                   // 它们所指向的对象也被自动删除了
```

由于复制 tr1::shared_ptr 的工作可以"如期进行"，所以在 auto_ptr 会出现另类的复制行为的地方，比如 STL 容器以及其它一些上下文中，这类指针能够安全地应用。

但是，请不要迷失方向。本条目并不是专门讲解 auto_ptr 和 tr1::shared_ptr 的，也不是讲解智能指针的。本条目的核心内容是使用对象管理资源的重要性。auto_ptr 和 tr1::shared_ptr 仅仅是这类对象的示例。（关于 tr1::shared_ptr 的更多信息，请参见条目 14、18 和 54。）

auto_ptr 和 tr1::shared_ptr 在析构函数中使用的都是 delete 语句，而不是 delete[]。（条目 16 中描述了二者的区别。）这就意味着对于动态分配的数组使用 auto_ptr 和 tr1::shared_ptr 不是一个好主意。但是遗憾的是，这样的代码会通过编译：

```
std::auto_ptr<std::string> aps(new std::string[10]);
                                   // 坏主意！
                                   // 这里将使用错误的删除格式
n
std::tr1::shared_ptr<int> spi(new int[1024]);
                                   // 同样的问题
```

你可能会很吃惊，因为在 C++中没有类似于 auto_ptr 和 tr1::shared_ptr 的方案来解决动态分配数组的问题，甚至 TR1 中也没有。这是因为 vector 和 string 通常都可以代替动态分配的数组。如果你仍然希望存在类似于 auto_ptr 和 tr1::shared_ptr 的数组类，请参见 Boost 的相关内容（见条目 55）。那儿会满足你需求：Boost 提供了 boost::scoped_array 和 boost::shared_array 来处理相关问题。

本条目中指引你使用对象来管理资源。如果你手动释放资源（比如使用 delete 而不是使用资源管理类），你就在做一些错事。诸如 auto_ptr 和 tr1::shared_ptr 等封装好的资源管理类通常可以让遵循本条目的建议变成一件很容易的事情，但是某些情况下，你的问题无法使用这些现成的类来解决，此时你便需要创建自己的资源管理类。但这并没有想象中那么难，但是确实需要你考虑一些细节问题。这些细节问题就是条目 14 和条目 15 的主题。

最后说一下，我必须指出 createInvestment 的原始指针返回类型存在着潜在的内存泄漏问题，因为调用者十分容易忘记在返回时调用 delete。（甚至在调用者使用 auto_ptr 或 tr1::shared_ptr 来运行 delete 时，他们仍然需要在一个智能指针对象中保存 createInvestment 的返回值。）解决这一问题需要改变 createInvestment 的接口，这是条目 18 的主题。

时刻牢记

- 为了避免资源泄漏，可以使用 RAII 对象，这类对象使用构造函数获取资源，析构函数释放资源。
- auto_ptr 或 tr1::shared_ptr 是两个常用并且实用的 RAII 类。通常情况下 tr1::shared_ptr 是更好的选择，因为它的复制行为更加直观。复制一个 auto_ptr 将会使其重设为 null。

条目14： 要注意资源管理类中的复制行为

条目 13 中介绍了“**资源获取即初始化**”（*Resource Acquisition Is Initialization*，简称 RAII）的概念，它是资源管理的中心内容。同时条目 13 中还使用 auto_ptr 和

tr1::shared_ptr 作为示例，描述了如何利用这一概念来管理堆上的资源。然而并不是所有的资源都分配于堆上，对于不分配于堆上的资源，诸如 auto_ptr 和 tr1::shared_ptr 这一类的智能指针并不适合于处理它们。这是千真万确的，你必须不时地自己动手，创建自己的资源管理类。

举例说，你正使用一个 C 版本的 API 所提供的 lock 和 unlock 函数来处理 Mutex 类型的互斥对象：

```
void lock(Mutex *pm);                  // 通过 pm 为互斥量上锁
void unlock(Mutex *pm);                // 为互斥量解锁
```

为了确保你曾上锁的互斥量都得到解锁，你应该自己编写一个类来管理互斥锁。这样的类的基本结构应遵循 RAII 的原理，那就是：资源在构造过程中获得，在析构过程中释放：

```
class Lock {
public:
  explicit Lock(Mutex *pm)
  : mutexPtr(pm)
  { lock(mutexPtr); }                  // 获取资源

  ~Lock() { unlock(mutexPtr); }        // 释放资源

private:
  Mutex *mutexPtr;
};
```

客户通过传统的 RAII 风格来使用 Lock 类：

```
Mutex m;                               // 定义互斥量以便使用
...
{                                      // 创建程序块用来定义临界区
 Lock ml(&m);                          // 为互斥量上锁
...                                    // 进行临界区操作
}                                      // 在程序块末尾互斥量将自动解锁
```

这样可以正常工作，但是如果复制一个 Lock 对象，将会发生些什么呢？

```
Lock ml1(&m);                    // 为 m 上锁

Lock ml2(ml1);                   // 把 ml1 复制给 ml2
                                 // 将会发生什么呢？
```

有一个问题是所有的 RAII 类创建者必须面对的，那就是：当复制一个 RAII 对象时应做些什么。以上是对于这个一般化问题的一个较具体的示例。大多数时候，以下四种可行的方案供你选择。

- **禁止复制。**在许多情况下，允许 RAII 被复制没有任何意义。比如对于 Lock 类来说就是这样，因为复制同步原型在大多数情况下都没有什么意义。当复制一个 RAII 类无意义时，你就应该禁止它。条目 6 中详细介绍了实现方法：将拷贝赋值运算符声明为私有的。对于 Lock 而言，应该是下面的情形：

```
class Lock: private Uncopyable {    // 防止复制 — 参见条目 6
public:
 ...                                // 同上
};
```

- **为潜在生成的资源进行引用计数。**有时，我们期望能保留对一个资源的所有权，直到其所涉及的最后一个对象被删除为止。在这种情况下，复制一个 RAII 对象将会添加一个引用资源对象的计数。这就是 tr1::shared_ptr 所使用的“复制”的含义。

  通常情况下，RAII 类可以通过包含一个 `tr1::shared_ptr` 数据成员来实现引用计数复制行为。举例说，如果 `Lock` 在设计时之初就期望使用引用计数，它可能会用 `tr1::shared_ptr<Mutex>`代替 `Mutex*`来作为 `mutexPtr` 的类型。但是不幸的是，`tr1::shared_ptr` 默认的行为是：当引用计数值变为零时，删除其所指向的内容，但这不是我们想要的。当一个 `Mutex` 用完时，我们希望对其进行的操作是解锁，而不是删除它。

  所幸的是，`tr1::shared_ptr` 允许指定一个“删除器”，它是一个函数或一个函数对象，用于在引用计数值为零时进行调用。（`auto_ptr` 并不包含这一特性，它**总是**删除它所指向的内容。）删除器是 `tr1::shared_ptr` 构造函数的第二个（可选的）参数，所以代码应该是这样的：

```
class Lock {
public:
  explicit Lock(Mutex *pm)      // 初始化 shared_ptr，参数为
  : mutexPtr(pm, unlock)        // 指向 Mutex 的指针和解锁函数

    lock(mutexPtr.get());       // 关于"get"的信息请参见条目 15
  }

private:
  std::tr1::shared_ptr<Mutex> mutexPtr;
};                              // 使用 shared_ptr 而不是原始指针
```

在本示例中，请注意 Lock 类不再声明析构函数。这是因为我们不再需要它了。条目 5 中介绍了类的析构函数（无论是编译器自动生成的还是用户自定义的）会自动为类的非静态数据成员进行析构。就像本示例中的 mutexPtr。然而，当互斥量的引用计数变为零时，mutexPtr 将会自动调用 tr1::shared_ptr 的删除器 unlock。（此时如果你为代码添加了一段注释，告诉人们你并没有忘记编写析构函数，你只是借助了默认的编译器行为。人们看了这样的注释思路会更清晰一些。他们会感激你的。）

- **复制潜在生成的资源。**一些时候，你可以在需要的情况下为资源复制出任意份数的副本，此时你需要一个资源管理类的唯一理由就是：确保每份副本在其工作完成之后得到释放。在这种情况下，复制资源管理对象的同时，也要复制出其所涉及的资源。也可以说，复制一个资源管理对象时，将进行“深度复制”。

  标准 string 类型的一些实现版本中，包含着一个指向堆内存的指针，这个指针所指向的就是字符串所保存的位置。这样的 string 对象包含着一个指向堆内存的指针。当一个 string 对象被复制时，将同时复制这一指针和其指向的内存。这样的 string 就进行了一次深度复制。

- **传递潜在生成资源的所有权。**在少数情况下，你可能需要确保仅仅有一个 RAII 对象引用了一个原始的资源，当复制这一 RAII 对象时，资源的所有权也将从源对象传递到目标对象。如同条目 13 中所解释的，这是通过 auto_ptr 所实现的“复制”的含义。

拷贝函数（拷贝构造函数和拷贝赋值运算符）可以由编译器自动生成，但是如果编译器自动生成版本无法满足你的需要（条目 5 中解释了 C++的默认行为），你就应该自己编写这些函数。在一些情况下，你可能还会需要支持这些函数的一般化的版本。这些版本将在条目 45 中介绍。

时刻牢记

- 由于复制一个 RAII 对象必须要同时复制其所管理的资源，因此资源的复制行为决定 RAII 对象的复制行为。
- RAII 类有两种一般性的复制行为：禁止复制和进行资源计数。同时其他的行为也是可能存在的。

条目15： 要为资源管理类提供对原始资源的访问权

资源管理类的特征是振奋人心的。它构筑起一道可靠的屏障，可有效地防止资源泄漏。能否预防资源泄漏是“系统的设计方案是否优异”的一个基本评判标准。在完美的世界里，你应该依靠资源管理类来完成所有的与资源交互的工作，而永远不要直接访问原始资源。然而世界并不是完美的。由于许多 API 会直接引用资源，因此除非你发誓不使用这样的 API（这样做显得太不实际了），否则，你必须绕过资源管理类，然后在需要的时候及时手工处理原始资源。

举例说，条目 13 中引入了下面的做法：使用诸如 `auto_ptr` 或者 `tr1::shared_ptr` 这样的智能指针来保存诸如 `createInvestment` 的工厂函数的返回值：

```
std::tr1::shared_ptr<Investment> pInv(createInvestment());
                                        // 来自条目 13
```

假设，当你使用 `Investment` 对象时，你需要一个这样的函数：

```
int daysHeld(const Investment *pi);  // 返回持有投资的天数
```

你可能希望这样来调用它：

```
int days = daysHeld(pInv);           // 错！
```

但是这段代码无法通过编译：因为 daysHeld 需要一个原始的 Investment*指针，但是你传递给它的对象的类型却是 tr1::shared_ptr<Investment>。

你需要一个渠道来将一个 RAII 类的对象（在上面的示例中是 tr1::shared_ptr）转变为它所包含的原始资源（比如说，原始的 Investment*）。这里实现这一转变有两个一般的方法：显式转换和隐式转换。

tr1::shared_ptr 和 auto_ptr 都提供了一个 get 成员函数来进行显式转换，也就是说，返回一个智能指针对象中的原始指针（的副本）：

```
int days = daysHeld(pInv.get());  // 工作正常，
                                  // 将 pInv 中的原始指针传递给 daysHeld
```

似乎所有的智能指针类，包括 tr1::shared_ptr 和 auto_ptr 等等，都会重载指针解析运算符（operator->和 operator*），这便使得你可以对原始指针进行隐式转换：

```
class Investment {                // 投资类型的层次结构中
                                  // 最为根基的类
public:
  bool isTaxFree() const;
  ...
};

Investment* createInvestment();   // 工厂函数

std::tr1::shared_ptr<Investment> pi1(createInvestment());
                                  // 使用 tr1::shared_ptr 管理资源
bool taxable1 = !(pi1->isTaxFree());
                                  // 通过 operator->访问资源
...

std::auto_ptr<Investment> pi2(createInvestment());
                                  // 使用 auto_ptr 管理资源
bool taxable2 = !((*pi2).isTaxFree());
                                  // 通过 operator*访问资源
...
```

由于某些时刻你需要获取一个 RAII 对象中的原始资源，所以一些 RAII 类的设计者使用了一个小手段来使系统正常运行，那就是：提供一个隐式转换函数。举例说，以下是一个 C 版本 API 中提供的处理字体的 RAII 类：

```
FontHandle getFont();               // 来自一个 C 版本 API
                                    // 省略参数表以简化代码

void releaseFont(FontHandle fh);    // 来自同一个 C 版本 API

class Font {                        // RAII 类
public:
  explicit Font(FontHandle fh)      // 通过传值获取资源
  : f(fh)                           // 因为该 C 版本 API 这样做
  {}

  ~Font() { releaseFont(f); }       // 释放资源

private:
  FontHandle f;                      // 原始的字体资源
};
```

假设这里有一个大型的与字体相关的 C 版本 API 通过 `FontHandle` 解决所有问题，那么把 `Font` 对象转换为 `FontHandle` 的操作将十分频繁。`Font` 类可以提供一个显式转换函数，比如 `get`：

```
class Font {
public:
  ...
  FontHandle get() const { return f; }
                                    // 进行显式转换的函数
  ...
};
```

遗憾的是，这样做使得客户在每次与这一 API 通信时都要调用一次 `get`：

```
void changeFontSize(FontHandle f, int newSize);
                                        // 来自该 C 语言 API
```

```
Font f(getFont());
int newFontSize;
...

changeFontSize(f.get(), newFontSize);   // 显式转换：从 Font 到 FontHandle
```

一些程序员可能会发现，由于使用这个类要求我们始终提供上述示例中的那种显式转换，这一点很糟糕，足够让他们拒绝使用这个类了。同时这一设计又增加了字体资源泄漏的可能性，这与 Font 类的设计初衷是完全相悖的。

有一个替代方案，让 Font 提供一个可隐式转换为 Fonthandle 的函数：

```
class Font {
public:
  ...
  operator FontHandle() const { return f; }     // 进行隐式转换的函数
  ...
};
```

这使得调用这一 C 版本 API 的工作变得简洁而且自然：

```
Font f(getFont());
int newFontSize;
...

changeFontSize(f, newFontSize);    // 隐式转换：从 Font 到 FontHandle
```

隐式转换会带来一定的负面效应：它会增加出错的可能。比如说，一个客户在一个需要 Font 的地方意外地创建了一个 FontHandle：

```
Font f1(getFont());
...
FontHandle f2 = f1;                 // 啊哦！本想复制一个 Font 对象，
                                    // 但却将 f1 隐式转换为其原始的
                                    // FontHandle，然后复制它
```

现在程序中有一个 FontHandle 资源正在由 Font 对象 f1 来管理，但是仍然可以通过 f2 直接访问 FontHandle 资源。这是很糟糕的。比如说，当 f1 被销毁时，字体就会被释放，f2 将无法被销毁。

是为 RAII 类提供显式转换为潜在资源的方法，还是允许隐式转换，上面两个问题的答案取决于 RAII 类设计用于完成的具体任务，及其被使用的具体环境。最好的设计方案应该遵循条目 18 的建议，让接口更容易被正确使用，而不易被误用。通常情况下，定义一个类似于 get 的显式转换函数是一个较好的途径，应为它可以使非故意类型转换的可能性降至最低。然而，一些时候使用隐式类型转换显得更加自然，人们更趋向于使用它。

你可能已经发现，让一个函数返回一个 RAII 类内部的原始资源是违背封装性原则的。的确是这样，乍看上去这简直就是设计灾难，但是它实际上并没有那么糟糕。RAII 类并不是用来封装什么的，它们是用来确保一些特别的操作能够得以执行的，那就是资源释放。如果需要，资源封装工作可以放在这一主要功能的最顶端，但是这并不是必需的。另外，一些 RAII 类结合了实现封装的严格性和原始资源封装的宽松性。比如 tr1::shared_ptr 对其引用计数机制进行了整体封装，但是它仍然为其所包含的原始指针提供了方便的访问方法。就像其它设计优秀的类一样，它隐藏了客户不需要关心的内容，但是它使得客户的确需要访问的部分对其可见。

时刻牢记

- API 通常需要访问原始资源，所以每个 RAII 类都应该提供一个途径来获取它所管理的资源。
- 访问可以通过显式转换或隐式转换来实现。一般情况下，显式转换更安全，而隐式转换对于客户来说更方便。

条目16： 互相关联的 new 和 delete 要使用同样的形式

下面的情景有什么不妥之处呢？

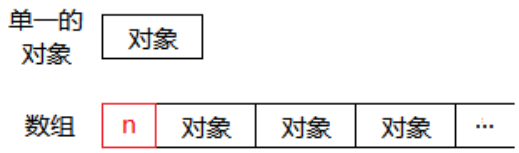
```
std::string *stringArray = new std::string[100];
...
```

```
delete stringArray;
```

一切似乎都按部就班，new 语句与 delete 相匹配。然而，这却是大错特错的。这段程序将出现未定义行为。最起码的是，由于该 stringArray 所指向的 100 个 string 对象中的 99 个没有被析构函数所析构，它们将很有可能不会被销毁。

当你使用了一个 new 语句时（也可以说，使用 new 动态创建了一个对象），将会发生两件事情。第一，分配内存（通过一个名为 operator new 的函数，参见条目 49 和 51）。第二，为这段内存调用一个或多个构造函数。当你使用了一个 delete 语句时，将会发生另外两件事情：第一，为分配的内存调用一个或多个析构函数。第二，释放内存（通过 operator delete 函数实现，参见条目 51）。delete 的关键问题是 :内存中存在多少需要删除的对象呢？答案取决于需要调用多少析构函数。

实际上，答案比这还要简单：你正在删除的指针是指向一个单独的对象，还是一组对象？这个问题很关键，因为为单个对象分配的内存与为一系列对象分配的内存在形式上有本质的不同。具体地说，为数组分配的内存通常要保存数组的大小，这就使得 delete 很容易知道需要调用多少次析构函数。为单个对象分配的内存则不保存这一信息。你可以将这一差别想象成下边图中的样子，其中 n 是数组的大小：



当然这仅仅是一个示例，并没有强制指标要求编译器以这种形式实现，尽管许多编译器确实是这样的。

当你对一个指针使用 delete 时，如何让 delete 知道这一指针是否存在数组信息呢？这里只有一种方法，那就是亲自告诉它。如果你在 delete 与指针名之间添加一对中括号，则 delete 便认为这一指针指向一个数组。否则将以单一对象处理。

```
std::string *stringPtr1 = new std::string;
std::string *stringPtr2 = new std::string[100];
...
```

```
delete stringPtr1;                    // 删除一个对象
delete [] stringPtr2;                 // 删除一个对象数组
```

如果你为 stringPtr1 使用“[]”时将会发生什么呢？我们说，会导致未定义行为。假设使用上面的内存分配形式，delete 将会读入一些内存信息，并且将其理解为数组的长度，然后便开始调用这么多的析构函数，此时 delete 不仅忽视了它正在操作的内存上保存的并不是数组，同时它“辛辛苦苦”析构的东西很有可能都不是它所能操作的类型。

如果你不为 stringPtr2 使用“[]”将会发生什么呢？同样会导致未定义行为。你可以看到由于它没有调用足够的析构函数，将造成内存泄漏。同时，对于内建数据类型，诸如 int 等，尽管它们没有析构函数，这个做法也将带来未定义行为（有时是有害的）。

这里的规则很简单：如果你在一个 new 语句中使用了[]，那么你必须在相关的 delete 语句中也使用[]。如果你在一个 new 语句中没有使用[]，那么在相关的 delete 语句中也不应使用[]。

有时候你会编写这样的类：它们包含用来动态分配内存的指针，并且提供多个构造函数。此时你需要时刻注意遵守上面的规则。在所有的构造函数中，你必须使用**一致格式的** new 来初始化指针成员。如果你不这样做，你怎么能知道析构函数中 delete 需要用什么样的格式呢？

如果你倾向于使用 typedef，那么这一规则同样值得你注意，因为它意味着：当你使用了 new 来创建 typedef 类型的对象时，至于应该使用 delete 语句的哪种形式，typedef 的作者必须事先做出说明。请看下边的示例：

```
typedef std::string AddressLines[4]    // 每个人的地址有 4 行，
                                       // 每行都是一个字符串
```

由于 AddressLines 是一个数组， 如果这样使用了 new：

```
std::string *pal = new AddressLines;   // 请注意“new AddressLines”
                                       // 返回一个 string*，
                                       // 与“new string[4]”完全一样
```

那么 delete 就必须使用**数组**的格式：

```
delete pal;                    // 将出现未定义行为！
delete [] pal;                 // 工作正常
```

为了避免此类混淆，请不要使用 typedef 来定义数组。这十分简单，因为 C++标准库（参见条目 54）中包含了 string 和 vector，使用这些模板可以摆脱动态分配数组的烦恼。比如说，在这里，AddressLines 可以定义为一个字符串的向量，也就是 vector<string>类型。

时刻牢记

- 如果你在一个 new 语句中使用了[]，那么你必须要在相关的 delete 语句中使用[]。如果你在 new 语句中没有使用[]，那么在相关的 delete 语句中一定不要出现[]。

条目17： 用智能指针存储由 new 创建的对象时要使用独立的语句

假设我们有一个函数用来展示处理的优先级，还有一个函数，它能够根据当前优先级的设置，为一个动态分配的 Widget 做一些处理：

```
int priority();
void processWidget(std::tr1::shared_ptr<Widget> pw, int priority);
```

一定要时刻记住“使用对象管理资源”（参见条目 13）。此处，processWidget 对其需要处理的动态分配的 Widget 使用了一个智能指针（在这里是一个 tr1::shared_ptr）。

下面是对 progressWidget 的一次调用：

```
processWidget(new Widget, priority());
```

请稍等，不要试图这样调用。这将不会通过编译。tr1::shared_ptr 的构造函数中包含一个原始指针，这个构造函数应为 explicit 的，于是便不存在从“new Widget”语句返回的原始指针到 processWidget 所需的 tr1::shared_ptr 的隐式转换。然而下边的代码将顺利通过编译：

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

看上去有些令人吃惊，尽管我们时时处处都使用对象来管理资源，但是这里还是有可能泄漏资源。了解其中的原由对深入理解是有一定启发性的。

在编译器能够生成对 processWidget 的调用之前，它必须对传入的参数进行预先的处理。第二个参数仅仅调用了一个函数 priority，但是第一个参数（“std::tr1::shared_ptr<Widget>(new Widget)”）包含两部分：

- 运行“new Widget”语句。
- 调用 tr1::shared_ptr 的构造函数。

因此，我们说在 processWidget 可以被调用之前，编译器必须自动生成代码来解决下面的三件事情：

- 调用 priority。
- 执行“new Widget”。
- 调用 tr1::shared_ptr 的构造函数。

C++编译器对于这三项任务完成的顺序要求得很宽松。（这一点与 Java 和 C#这类语言很不一样，这类语言中的函数参数总是以一个特定的顺序得到预处理。）由于“new Widget”语句运行的结果是一个参数的形式传递给 tr1::shared_ptr 的构造函数的，因此它必须在 tr1::shared_ptr 的构造函数被调用之前得到执行。但是调用 priority 的工作可以放到第一，第二，也可以放在最后。如果编译器决定第二个处理它（这样可以使编译器生成的代码更高效），我们就会得到这样的执行序列：

1. 执行“new Widget”。
2. 调用 priority。
3. 调用 tr1::shared_ptr 的构造函数。

但是请想象一下：如果调用 priority 时抛出了一个异常的话，将会发生些什么。在这种情况下，由“new Widget”返回的指针将会丢失。这是因为这一指针并不会

保存在 tr1::share_ptr 中，然而我们原本还期望利用 tr1::shared_ptr 来避免资源泄露。这种情况下调用 processWidget 可能会造成资源泄漏。这是因为：在资源被创建（通过 new Widget）以后和将这个资源转交给一个资源管理对象之前的这段时间内，有产生异常的可能。

防止这类问题发生的办法很简单：使用单独的语句，创建 Widget 并将其存入一个智能指针，然后将这个智能指针传递给 processWidget：

```
std::tr1::shared_ptr<Widget> pw(new Widget);
                                    // 在一个单独的语句中创建 Widget
                                    // 将其存入一个智能指针

processWidget(pw, priority());      // 这样调用就不会泄漏了。
```

这样是可行的，因为编译器为**多行**语句安排执行顺序要比**单一**的语句时严格得多。由于这段改进的代码中，“new Widget”语句以及 tr1::shared_ptr 的构造函数将在单独的语句中得到调用，而对 priority 的调用在另一个单独的语句中，所以编译器就没有机会将对 priority 的调用挪动到“new Widget”语句和 tr1::shared_ptr 的构造函数之间了。

时刻牢记

- 在智能指针中的由 new 创建的对象要在单独的语句中保存。如果不这样做，你的程序会在抛出异常时发生资源泄漏。

第四章. 设计规划与声明方式

软件设计方案是用来帮助规划软件功能的，通常，它在一开始是一系列优秀的思想，但最终都会被具体化，直至其成为可用于开发的一系列具体接口。之后，这些接口必须转换为 C++的声明。本章中，我们将解决“如何设计和声明优秀的 C++接口”这一问题。我们以下面的建议开始（它可能是设计接口时要注意的最为重要的一点）：接口应该易于正确使用，而不易被误用。这引出了一系列更为具体的建议，其中我们将讨论一些十分宽泛的议题，包括正确性、效率、封装、可维护性、可扩展性，还有你需要遵循的惯例。

本章并不期望对于设计优秀接口的建议做到包罗万象，取而代之的是精选出一部分最为重要的内容。各种错误和问题对于类/函数/模板的设计者来说是家常便饭，本章就是用来时刻警示你远离这些常见的错误，并为你提供这些问题的解决方法。

条目18： 要让接口易于正确使用，而不易被误用

C++中到处充满了接口。函数接口、类接口、模板接口，等等。每个接口都是实现客户与你的代码相交互的一种手段。假设你的客户都是完全理性的，他们致力于更优秀的完成当前项目，他们便会十分看重你的接口是否能够正确使用。这样一来，如果你的接口中的任意一个被他们误用了，那么这个接口便成了这一错误的“罪魁祸首”。在理想状态下，如果客户尝试使用一个接口，但是没有达到预期的效果，那么代码则不应通过编译。反之，如果代码通过了编译，则运行结果必须要符合客户的需求。

开发中我们应做到让接口更易于正确使用而不易被误用，这需要你考虑到客户会犯的各种错误。请参见下边的示例，假设你正在设计一个表示日期时间的类的构造函数：

```
class Date {
public:
  Date(int month, int day, int year);
  ...
};
```

乍一看，这一接口设计得很合理（至少在美国很合理），但是当客户面对这样的接口时，很容易犯下两种错误。第一，他们可能会使用错误的传参顺序：

```
Date d(30, 3, 1995);              // 啊哦，应该是“3, 30”而不是“30, 3”
```

第二，他们可能会传进一个无效的月份或日期：

```
Date d(3, 40, 1995);              // 啊哦，应该是“3, 30”而不是“3, 40”
```

（这一示例看上去有些愚蠢，但是不要忘了，在键盘上 3 和 4 是紧挨着的。这种“擦肩而过”的错误在现实中并不少见）

客户犯下的许多错误是可以通过引入新类型来避免的。实际上，对于防止不合要求的代码通过编译，类型系统是你最得力的助手。在上述情况下，我们可以引入几个简单的“包装类型”来区分日期、月份、和年份，然后再在 Date 的构造函数中使用这些类型：

```
struct Day {
  explicit Day(int d) : val(d) {}
  int val;
};

struct Month {
  explicit Month(int m) : val(m) {}
  int val;
};

struct Year {
  explicit Year(int y) : val(y) {}
  int val;
};

class Date {
public:
 Date(const Month& m, const Day& d, const Year& y);
 ...
};
```

```
Date d(30, 3, 1995);                          // 报错！类型错误
Date d(Day(30), Month(3), Year(1995));        // 报错！类型错误
Date d(Month(3), Day(30), Year(1995));        // OK，类型正确
```

我们可以改善上边应用结构体的简单思路，让 Day、Month、Year 变得“羽翼丰满”，从而可以提供完善的数据封装性（参见条目 22）。但是即使是结构体也足以说服我们：适时引入新的类型可以十分有效地防止接口误用。

只要你在恰当的地方使用了恰当的类型，你便可以合理地限制这些类型的值。比如说，一年有 12 个月，所以 Month 类型应该能够反映出这一点。一个途径是使用枚举类型来表示月份，但是枚举类型并不总能达到我们对于类型安全的需求。比如说，枚举类型可以像 int 一样使用（参见条目 2）。一个更安全的解决方法是：预先定义好所有有效 Month 值的集合：

```
class Month {
public:
  static Month Jan() { return Month(1); }   // 用来返回所有有效月份值的函数；
  static Month Feb() { return Month(2); }   // 下文中解释为何用函数而不是对象
  ...
  static Month Dec() { return Month(12); }

  ...                                       // 其他成员函数

private:
  explicit Month(int m);                    // 防止创建新的月份值
  ...                                       // 与月份相关的数据
};

Date d(Month::Mar(), Day(30), Year(1995));
```

如果上面代码中使用函数来代替具体月份的思路让你感到奇怪，那么可能是由于你已经忘记了声明非局部静态对象可能会带来可靠性问题。条目 4 可以唤醒你的记忆。

为防止客户犯下类似的错误，我们还可以采用另一个途径，那就是严格限制一个类型可以做的事情。加强限制的一个常用的手段就是添加 const 属性。比如说，条目 3 中曾解释过，const 是如何通过限定 operator*的返回值，从而防止客户对用户定义类型犯下以下的错误的：

```
if (a * b = c) ...                    // 啊哦，本来是想进行一次比较！
```

实际上，这仅仅是针对“让接口易于正确使用，而不易被误用”另一条一般性建议的一个表现形式，这条建议是：除非有更好的理由阻止你这样做，否则你应该保证你所创建类型的行为与内建数据类型保持一致。因为客户已经清楚 int 的行为，所以只要是合情合理，你就应该力求使你的类拥有与 int 一致的行为。比如说，如果 a 和 b 是 int 类型，那么为 a*b 赋值就是不合法的。所以除非你有好的理由拒绝这一规定，否则你自己创建的类型也应该将这一行为界定为不合法。当你举棋不定时，就让你的类型的行为与 int 保持一致。

设计接口时应避免与内建数据类型之间存在不必要的不兼容问题，这样做的真正目的是保持各类接口行为的一致性。很少有特征能像一致性这样，可以让接口如此易于正确使用；同时，也很少有特征能像不一致性那样，可以让接口变得那般糟糕。STL 容器的接口大体上（但并不完美）是一致的，这就使得它们更易于使用。比如说每个 STL 容器都有一个名为 size 成员函数，它可以告诉我们当前这一容器中容纳了多少对象。这一点与 Java 和.NET 是不同的，Java 中使用 length **属性**来表示数组的长度，length **方法**来表示字符串的长度，以及 size 方法来表示 List 的大小。而.NET 中的 Array 拥有一个叫做 Length 的属性，而 ArrayList 中功能相类似的属性则叫做 Count。一些开发人员认为，集成开发环境（IDE）使得这类不一致性问题变得不那么重要，但是实际上他们想错了。不一致性问题会给开发人员带来无穷尽的烦恼，没有哪个 IDE 是能够完美解决这些问题的。

任何接口都需要客户记忆一些易发生错误的内容，这是因为客户可能会把这些东西搞砸。比如说，条目 13 中曾引入一个工厂函数来返回一个指向 Investment 层中动态分配对象的指针：

```
Investment* createInvestment();    // 来自条目 13，省略参数表以简化代码
```

为防止资源泄漏，由 createInvestment 返回的指针在最后必须被删除，但是这将会给客户留下至少两个犯错误的机会：忘记删除指针、多于一次删除同一指针。

条目 13 中介绍了客户如何将 createInvestment 的返回值保存在诸如 auto_ptr 或 tr1::shared_ptr 这样的智能指针中，然后让智能指针担负起调用 delete 的责任。但是如果客户忘记了使用智能指针，这该怎么办呢？通常情况下，更好的接口的设计方案是：让工厂函数返回一个智能指针，在一开始就不给问题任何发生的机会。

```
std::tr1::shared_ptr<Investment> createInvestment();
```

这样便可以从根本上强制客户使用 tr1::shared_ptr 来存储返回值，这一做法基本上可以排除“忘记删除当前不再有用的 Investment 对象”的可能。

事实上，返回 tr1::shared_ptr 让接口设计人员能够防止与资源释放相关的客户端错误，这是因为在创建 tr1::shared_ptr 智能指针时，允许存在一个与当前智能指针相绑定的资源释放函数（即一个“删除器”），而 auto_ptr 没有这一功能。（参见条目 14）

假设客户从 createInvestment 中得到了一个 Investment*指针，在进行删除操作时，我们期望这一客户将这个指针传给一个名为 getRidOfInvestment 的函数，而不是使用 delete。在这里，如果客户会使用错误的资源析构机制（也就是使用 delete 而不是 getRidOfInvestment），那么这样的接口就带来了新的客户端错误。实现 createInvestment 的程序员可以通过返回一个绑定 getRidOfInvestment 作为“删除器”的 tr1::shared_ptr 来预防此类错误。

tr1::shared_ptr 提供了一个拥有两个参数的构造函数，这两个参数即：需要管理的指针，以及当引用计数值为零时需要调用的删除器。这使得我们可以创建使用 getRidOfInvestment 作为“删除器”的空 tr1::shared_ptr，请看下面的做法：

```
std::tr1::shared_ptr<Investment> pInv(0, getRidOfInvestment);
                                    // 尝试创建一个 null 的 shared_ptr，
                                    // 并且让其包含一个自定义的删除器；
                                    // 这样的代码无法通过编译
```

然而，这并不是合法的 C++语法。tr1::shared_ptr 的构造函数的第一个参数必须是一个**指针**，而 0 则不是，它是一个 int 值。的确，数字可以**当做**指针使用，

但是这种情况下该做法并不值得推荐，tr1::shared_ptr 的第一个参数必须是一个实际的指针。通过一次转型可以解决这一问题：

```
std::tr1::shared_ptr<Investment>
  pInv(static_cast<Investment*>(0), getRidOfInvestment);
                                    // 创建一个 null 的 shared_ptr，
                                    // 并且让其包含一个自定义的删除器；
                                    // static_cast 的更多信息参见条目 27
```

上面的代码意味着，在实现 createInvestment 时，可让其返回一个“绑定了 getRidOfInvestment 删除器的 tr1::shared_ptr”：

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  std::tr1::shared_ptr<Investment>
    retVal(static_cast<Investment*>(0),  getRidOfInvestment);

  retVal = ... ;                    // 让 retVal 指向恰当的对象
  return retVal;
}
```

当然，如果在创建 pInv 之前就确定了其所管理的原始指针，那么，比起“将 pInv 初始化为空值然后对其赋值”而言，“将原始指针传递给 pInv 的构造函数”的方法更理想些。这是为什么呢？详情请参见条目 26。

tr1::shared_ptr 可以自动为每个指针预留一个删除器，它们可以排除另一类潜在的客户端错误，即所谓的“跨 DLL 问题”，这是 tr1::shared_ptr 的一项尤为显著的优点。如果一个动态链接库（DLL）中使用 new 创建了一个对象，而在另一个 DLL 中这个对象被 delete 语句删除了，那么此时将会引发“跨 DLL 问题”。在许多平台上，此类跨 DLL 的“new/delete 对”将导致运行时错误。tr1::shared_ptr 可以防止此类问题发生，因为如果创建了一个 tr1::shared_ptr，它的默认删除器将在同一个 DLL 中使用 delete。举例说，如果 Stock 继承自 Investment，同时 createInvestment 是这样实现的：

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  return std::tr1::shared_ptr<Investment>(new Stock);
```

```
}
```

那么返回的 tr1::shared_ptr 能够在各 DLL 文件中自由穿梭，而不用考虑跨 DLL 问题。这一指向 Stock 的 tr1::shared_ptr 会始终追踪这一事件：当 Stock 的引用计数值为零时，需要使用哪一个 DLL 的 delete 语句来删除它。

本条目讲解的主要内容是如何让接口更加易于正确使用，而不易被误用，而不是 tr1::shared_ptr，但是 tr1::shared_ptr 对于避免此类客户端错误却是一个不可多得的好工具，学会使用它是值得的。tr1::shared_ptr 最为通用的实现来自 Boost（参见条目 55）。Boost 中的 shared_ptr 有两个原始指针那么大，它在存储计数信息和删除器相关的数据时会使用动态分配的内存，在进行删除器调用时会使用虚函数，对于其识别为多线程的应用程序，在修改引用计数时会引入线程同步的开销。(你也可以通过定义一个预处理符号来禁用多线程。) 简言之：它比原始指针的体积更大，执行速度更慢，并且使用辅助动态内存。在许多应用中，这些额外的运行时开销是微不足道的，但是它可以显著降低每个客户出错的可能，这一点绝对是振奋人心的。

时刻牢记

- 优秀的接口应该易于正确使用，而不易误用。你应该力争让你所有的接口都具备这一特征。
- 增加易用性的方法包括：让接口保持一致性，让代码与内建数据类型保持行为上的兼容性。
- 防止错误发生的方法包括：创建新的数据类型，严格限定类型的操作，约束对象的值，主动管理资源以消除客户的资源管理职责。
- tr1::shared_ptr 支持自定义的删除功能。可以防止“跨 DLL 问题”，可以用于自动解开互斥锁（参见条目 14）。

条目19： 要像设计类型一样设计 class

与其它的面向对象编程语言类似，在 C++中，定义一个新的 class 即定义了一个新的类型。一个 C++开发者的职业生涯的大多数时间都将用在“不断丰富充实他们

的类型系统"上。这意味着他不仅仅是一个 class 的设计者，更是一个**类型**的设计者。函数和运算符重载、内存的分配和释放控制、对象初始化和终止定义——一切都由设计人员手工完成。我们知道，语言设计人员在设计内建数据类型时倾注了大量心血，而一个 class 设计人员也要花费同样的精力。

能否设计出优秀的 class 对于设计人员来说是一项严峻的考验，因为设计出好的数据类型本身就是一项艰巨的任务。优秀的类型拥有自然的语法、直观的语义，并且还有一套或多套高效的实现。在 C++中，如果定义 class 的工作做得一团糟，那么期望达到上面的目标就是天方夜谭。甚至 class 的成员函数的声明方式也会影响到它的性能。

那么，如何把 class 设计得更高效呢？首先，你必须要了解你所面对的问题。几乎所有的 class 设计都将面对下面的问题，它们的答案可以对设计起到一定的约束作用：

- **新类型的对象应如何创建和删除？**class 中与之相关的函数包括：构造函数和析构函数，以及 class 中其它的内存分配和释放函数（operator new、operator new[]、operator delete、operator delete[]，参见第八章）。如果你自己手动编写它们，这个问题的解决方式将会影响到这些函数。
- **对象初始化与对象赋值有怎样的不同？**这个问题的答案决定着构造函数与赋值运算符之间的区别。不要混淆初始化和赋值的概念，这一点很重要，因为二者所面对的函数调用类型是不同的。
- **新类型在通过传值方式传递对象时意味着什么？**请牢记，一个类型是通过拷贝构造函数来定义传值操作的实现方式的。
- **新类型对合法数值有哪些限制？**通常情况下，对于某个 class 的数据成员而言，只有一些特定的数值组合是合法的。这些组合决定了 class 应遵循哪些定律。而这些定律又决定了在数据成员中你应该进行哪些错误检查，尤其是构造函数、赋值运算符、以及"设定"函数（即 setter）。它们还会影响到函数会抛出什么样的异常，同时在某些情况下还有可能影响到函数所抛出异常的细节。

- **新类型是否适用于继承？**如果新的 class 由现有的 class 继承而来，那么新的 class 应遵循现有 class（即父类）设计方案的限制。尤其是要确定父类的成员函数是否为虚函数（参见条目 34 和 36）。如果期望让其它的 class 可以继承当前的 class，就需要考虑当前 class 的成员函数是否应为虚函数，尤其是它的析构函数（参见条目 7）。

- **新类型允许进行哪些类型转换？**新的类型存在于各式各样的类型之间，那么是否应该提供新类型与其它类型的类型转换功能呢？如果你期望让 T1 的一个对象将类型隐式转换为 T2。可以通过在 T1 类中放置一个类型转换函数（比如 operator T2），或者在 T2 类中放置一个有单一参数的非 explicit 构造函数。如果你期望 T1 仅允许显式类型转换，就需要编写函数来执行这一转换，但是这一函数不应是类型转换运算符，也不应是单一参数的非 explicit 构造函数。（条目 15 中有隐式/显式转换函数的示例。）

- **哪些运算符和函数对新类型是有意义的？**这个问题的答案取决于你会为你的 class 声明哪些函数。一些函数将成为成员函数，另一些则不是（参见条目 23、24、46）。

- **应明确拒绝哪些标准函数？**通过将它们声明为 private 的可达到这一目的（参见条目 6）。

- **谁可以访问新类型中的数据成员？**这一问题可以帮助我们确定哪些成员应为 public 的，哪些是 protected 的，以及哪些是 private 的。同时，也可以帮助我们确定哪些 class 和/或函数应该是友元，还有嵌套的 class 是否有意义。

- **新类型中有哪些“未声明的接口”？**如果你充分考虑了新类型中性能、异常安全（参见条目 29）、资源使用（比如互斥锁、动态内存）等问题，系统将许诺给你什么呢？我们说你在这些领域所作出的努力，将确保你的 class 的实现中相应的约束条件能够得以严格实施。

- **新类型有多通用？**可能你想做的并不仅仅是定义一个新类型。而是定义一族新类型。如果真是这样，需要你定义的就不是一个新的 class 了，你需要定义一个新的类模板（class template）。

- **你真的需要一个新类型吗？**如果你创建新的派生类仅仅为了为现有的类添加新的功能，那么通过简单地定义一个或多个非成员函数或者模板可能会更好的达到目标。

完整地回答以上的问题列表并不是一件简单的事情，因此定义高效的 class 就是一项严峻的挑战。然而，如果成功完成了这一挑战，那么由用户自定义的 class 生成的类型至少可以像内建数据类型一样好用。一切都是值得的。

时刻牢记

- class 设计就是类型的设计。在定义一个新的类型之前，要确保将本条目讨论的所有问题考虑周全。

条目20： 传参时要多用“引用常量”，少用传值

默认情况下，C++为函数传入和传出对象是采用传值方式的（这是由 C 语言继承而来的特征）。除非你明确使用其他方法，函数的形式参数总会通过复制实在参数的**副本**来创建，并且，函数的调用者得到的也是函数返回值的一个**副本**。这些副本是由对象的拷贝构造函数创建的。这使得“传值”成为一项代价十分昂贵的操作。请观察下边的示例中类的层次结构：

```
class Person {
public:
  Person();                          // 省略参数表以简化代码
  virtual ~Person();                 // 条目 7 解释了它为什么是虚函数
  ...

private:
  std::string name;
  std::string address;
};

class Student: public Person {
public:
  Student();                         // 再次省略参数表
```

```
  virtual ~Student();
  ...

private:
  std::string schoolName;
  std::string schoolAddress;
};
```

请观察下面的代码，这里我们调用一个名为 validateStudent 的函数，通过为这一函数传进一个 Student 类型的参数（传值方式），它将返回这一学生的身份是否合法：

```
bool validateStudent(Student s);          // 通过传值方式接受一个 Student 对象

Student plato;                            // 柏拉图是苏格拉底的学生

bool platoIsOK = validateStudent(plato); // 调用这一函数
```

在这个函数被调用时将会发生些什么呢？

很显然地，在这一时刻，通过调用 Student 的拷贝构造函数，可以将这一函数的 s 参数初始化为 plato 的值。同样显然的是，s 在 validateStudent 返回的时候将被销毁。所以这一函数中传参的开销就是调用一次 Student 的拷贝构造函数和一次 Student 的析构函数。

但是上边的分析仅仅是冰山一角。一个 Student 对象包含两个 string 对象，所以每当你构造一个 Student 对象时，你都必须构造两个 string 对象。同时，由于 Student 类是从 Person 类继承而来，所以在每次构造 Student 对象时，你都必须再构造一个 Person 对象。一个 Person 对象又包含两个额外的 string 对象，所以每次对 Person 的构造还要进行额外的两次 string 的构造。最后的结果是，通过传值方式传递一个 Student 对象会引入以下几个操作：调用一次 Student 的拷贝构造函数，调用一次 Person 的拷贝构造函数，调用四次 string 的拷贝构造函数。在 Student 的这一副本被销毁时，相应的每次构造函数调用都对应着一次析构函数的调用。因此我们看到：通过传值方式传递一个 Student 对象总体的开销究竟有多大？竟达到了六次构造函数和六次析构函数的调用！

下面向你介绍正确的方法，这一方法才会使函数拥有期望的行为。毕竟你**期望**的是所有对象以可靠的方式进行初始化和销毁。与此同时，如果可以绕过所有这些构造和析构操作将是件很惬意的事情。这个方法就是：通过引用常量传递参数：

```
bool validateStudent(const Student& s);
```

这样做效率会提高很多：由于不会创建新的对象，所以就不会存在构造函数或析构函数的调用。改进的参数表中的 const 是十分重要的。由于早先版本的 validateStudent 通过传值方式接收 Student 参数，所以调用者了解：无论函数对于传入的 Student 对象进行什么样的操作，都不会对原对象造成任何影响，validateStudent 仅仅会对对象的**副本**进行修改。而改进版本中 Student 对象是以引用形式传入的，有必要将其声明为 const 的，因为如果不这样，调用者就需要关心传入 validateStudent 的 Student 对象有可能会被修改。

通过引用传参也可以避免“**截断问题**”。当一个派生类的对象以一个基类对象的形式传递（传值方式）时，基类的拷贝构造函数就会被调用，此时，这一对象的独有特征——使它区别于基类对象的特征会被“截掉”。剩下的只是一个简单的基类对象，这并不奇怪，因为它是由基类构造函数创建的。这肯定不是你想要的。请看下边的示例，假设你正在使用一组类来实现一个图形窗口系统：

```
class Window {
public:
  ...
  std::string name() const;         // 返回窗口的名字
  virtual void display() const;     // 绘制窗口和内容
};

class WindowWithScrollBars: public Window {
public:
  ...
  virtual void display() const;
};
```

所有的 Window 对象都有一个名字，可以通过 name 函数取得。所有的窗口都可以被显示出来，可以通过调用 display 实现。display 是虚函数，这一事实告诉我

们，简单基类 Window 的对象与派生出的 WindowWithScrollBars 对象的显示方式是不一样的。（参见条目 34 和 36）

现在，假设你期望编写一个函数来打印出当前窗口的名字然后显示这一窗口。下面是**错误**的实现方法：

```
void printNameAndDisplay(Window w) // 错误！参数传递的对象将被截断！
{
  std::cout << w.name();
  w.display();
}
```

考虑一下当你将一个 WindowWithScrollBars 对象传入这个函数时将会发生些什么：

```
WindowWithScrollBars wwsb;

printNameAndDisplay(wwsb);
```

参数 w 将被构造为一个 Window 对象——还记得么？它是通过传值方式传入的。这里，使 wwsb 具体化的独有信息将被截掉。无论传入函数的对象的具体类型是什么，在 printNameAndDisplay 的内部，w 将总保有一个 Window 类的对象的身份（因为它**本身就是**一个 Window 的对象）。特别地，在 printNameAndDisplay 内部对 display 的调用**总是** Window::display，而永远不会是 WindowWithScrollBars::display。

解决截断问题的方法是：通过引用常量传参：

```
void printNameAndDisplay(const Window& w)
{                                   // 工作正常，参数将不会被截断。
  std::cout << w.name();
  w.display();
}
```

现在 w 的类型就是传入窗口对象的精确类型。

揭开 C++编译器的面纱，你将会发现引用通常情况下是以指针的形式实现的，所以通过引用传递通常意味着实际上是在传递一个指针。因此，如果传递一个内建数据

类型的对象（比如 int），传值会被传递引用更为高效。那么，对于内建数据类型，当你在传值和传递常量引用之间徘徊时，传值方式不失为一个更好的选择。迭代器和 STL 中的函数对象也是如此，这是因为它们设计的初衷就是能够更适于传值，这是 C++的惯例。迭代器和函数对象的设计人员有责任考虑复制时的效率问题和截断问题。（这也是一个“使用哪种规则，取决于当前使用哪一部份的 C++”的例子，参见条目 1）

内建数据类型体积较小，所以一些人得出这样的结论：所有体积较小的类型都适合使用传值，即使它们是用户自定义的。这是一个不可靠的推理。仅仅通过一个对象体积小并不能判定调用它的拷贝构造函数的代价就很低。许多对象——包括大多数 STL 容器——其中仅仅包含一个指针和很少量的其它内容，但是复制此类对象的同时，它所指向的所有内容都需要复制。这将付出**十分**高昂的代价。

即使体积较小的对象的拷贝构造函数不会带来巨大的开销，它也会引入性能问题。一些编译器对内建数据类型和用户自定义数据类型是分别对待的，即使它们的表示方式完全相同。比如说一些编译器很乐意将一个单纯的 double 值放入寄存器中，这是语言的常规；但将一个仅包含一个 double 值的对象放入寄存器时，编译器就会报错了。当你遇到这种事情时，你可以使用引用传递这类对象，因为编译器此时一定会将指针（引用的具体实现）放入寄存器中。

对于“小型的用户自定义数据类型不适用于传值方式”还有一个理由，那就是：作为用户自定义类型，它们的大小可能会改变。现在很小的类型在未来的版本中可能会变得很大，这是因为它的内部实现方式可能会改变。即使是你更改了 C++语言的具体实现都可能会影响到类型的大小。比如，在我编写上面的示例的时候，一些对标准库实现中 string 的大小竟然达到了另一些的**七倍**。

总体上讲，只有内建数据类型、STL 迭代器和函数对象类型适用于传值方式。对于所有其它的类型，都应该遵循本条款中的建议：尽量使用引用常量传参，而不是传值。

时刻牢记

- 尽量使用引用常量传参，而不是传值方式。因为一般情况下传引用更高效，而且可以避免“截断问题”。

- 对于内建数据类型、STL 迭代和函数对象类型，这一规则就不适用了，对它们来说通常传值方式更实用。

条目21： 在必须返回一个对象时，不要去尝试返回一个引用

一旦程序员把注意力都转向了对象传值方式隐含的效率问题（参见条目 20）时，许多人都变成了极端的“改革运动者”，他们对传值方法采取斩草除根的态度，不屈不挠的追随着一个纯粹的传递引用的世界，与此同时，他们也无可避免的犯下了一个致命的错误：有时候传递的引用所指向的对象并不存在。这决不是一件好事情。

请看下面的示例，其中的 Rational 类用来表示有理数，还有一个函数用来计算两个有理数的乘积：

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
                                    // 条目 24 中解释了为什么这里的构造函数
                                    // 没有声明为 explicit。
  ...
private:
  int n, d;                         // 分子（n）和分母（d）

friend const Rational
                                    operator*(const Rational& lhs, const
Rational& rhs);
                                    // 条目 3 中解释了为什么返回值是 const 的。
};
```

这一版本的 operator*通过传值方式返回一个对象，如果你不去考虑这一对象在构造和析构过程中的开销，那么你就是在逃避你的专业职责。如果你并不是非得要为这样的对象付出代价，那么你大可不必那样做。现在问题就是：你必须付出这一代价吗？

好的，如果此时你可以返回一个引用作为替代品，那么就不需要了。但是请记住，一个引用仅仅是一个**名字**，它是一个**已存在**对象的别名。当你看到一个引用的声明

时，你应该立刻问一下你自己：它的另一个名字是什么，因为一个引用所指向的内容必定有它自己的名字。于是对于上面的 operator*而言，如果它返回一个引用，那么它所引用的必须是一个已存在的 Rational 对象，这个对象中包含着需要进行乘法操作那两个对象的乘积。

这里我们没有理由期望在调用 operator*之前这一对象必须存在。也就是说，如果你这样做了：

```
Rational a(1, 2);                    // a = 1/2
Rational b(3, 5);                    // b = 3/5

Rational c = a * b;                  // c 的值应该为 3/10
```

这里并没有理由期待此处已经存在一个值为 3/10 的有理数。其实并不是这样的，如果 operator*返回一个指向这类数值的引用，那么它必须要自己创建这个数字。

一个函数只能以两种方式创建新的对象：在栈上或在堆上。在栈上创建一个新对象是通过定义一个局部变量完成的。应用这一策略时，你可能会以这种方式编写 operator*：

```
const Rational& operator*(const Rational& lhs, const Rational& rhs)
                                     // 警告！错误的代码
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

你完全可以拒绝这样的实现方法，因为你的目标是防止对构造函数的调用，但是此时 result 会像其它对象一样被初始化。一个更严重的问题是：这个函数会返回一个指向 result 的引用，但是 result 是一个局部对象，而局部对象在函数退出时就会被销毁。那么，这一版本的 operator*，并不会返回一个指向 Rational 的引用，它返回的引用指向一个“前 Rational”，一个“空寂的、散发着霉气的、开始腐烂的尸体”，它**曾经是**一个 Rational 对象，但它现在与 Rational 已经毫无关系，因为它已经被销毁了。对于所有的调用者而言，只要**稍稍触及**这一函数的返回值，都会遭遇到无尽的未定义行为。事实上，任何返回局部对象引用的函数都是灾难性的。（任何返回指向局部对象的指针的函数也是如此。）

现在，让我们考虑下面做法的可行性：在堆上创建一个对象，然后返回一个指向它的引用。由于保存于堆上的对象由 new 来创建，因此你可能会这样编写基于堆的 operator*：

```
const Rational& operator*(const Rational& lhs, const Rational& rhs)
                                    // 警告！更多的错误代码！
{
  Rational *result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

好的，此时仍然需要付出调用构造函数的代价，这是因为通过 new 分配的内存要通过调用一个合适的构造函数来初始化，但是现在你面临这另一个问题：谁来确保与 new 相对应的 delete 的执行呢？

即使调用者十分认真负责并且抱有良好的初衷，他们也无法保证下面这样合理的使用场景下不会出现内存泄漏：

```
Rational w, x, y, z;

w = x * y * z;                      // 等价于 operator*(operator*(x, y), z)
```

这里，在一个语句中存在着两次对 operator*的调用，于是存在两次 new 操作有待于使用 delete 来清除。但是又没有任何理由要求 operator*的客户来进行这一操作，这是因为对 operator*的调用返回了一个引用，没有理由要求客户去取得隐藏在这一引用背后的指针。这势必会造成资源泄漏。

但是，也许你注意到了，栈方案与堆方案都面临着同一个问题：它们都需要为 operator*的每一个返回值调用一次构造函数。也许你能够回忆起我们最初的目的就是避免像此类构造函数调用。也许你认为你知道某种方法来将此类构造函数调用的次数降低到仅有一次。也许你想到了下面的实现方法：让 operator*返回一个指向一个**静态的** Rational 对象的引用（这一静态对象放置于函数的**内部**）：

```
const Rational& operator*(const Rational& lhs, const Rational& rhs)
                                    // 警告！更多更多的错误代码！
{
  static Rational result;           // 用来作为返回值的静态对象
```

```
  result = ... ;                    // 将 lhs 与 rhs 相乘，
                                    // 并将乘积存入 result
  return result;
}
```

与其它引入静态对象的设计方法一样，这种方法很显著的提高了线程的安全性，但是这却带来了更明显的缺陷。下面的客户端代码是无懈可击的，但是上文中的设计会使其暴露出更深层次的缺陷：

```
bool operator==(const Rational& lhs, const Rational& rhs);
                                    // 为有理数作比较的 operator==
Rational a, b, c, d;

...
if ((a * b) == (c * d))  {
    当乘积相等时，执行恰当的操作;
} else   {
    当乘积不相等时，执行恰当的操作;
}
```

猜猜会发深什么？无论 a、b、c 或 d 取什么值，表达式`((a*b) == (c*d))`的值**永远**为 `true`。

我们为上面代码中的判断语句更换为函数的形式，这个问题就更加浅显了：

```
if (operator==(operator*(a, b), operator*(c, d)))
```

请注意，在调用 `operator==`时，已经存在了**两次**活动的 `operator*`调用，每次调用时都回返回一个指向 `operator*`内部的静态 `Rational` 对象的引用。于是编译器将要求 `operator==`去将 `operator*`内部的静态 `Rational` 对象与自身相比较。如果结果不相等，才是让人吃惊的事情。

上面的内容似乎已经足够让你确信：为类似于 `operator*`这样的函数返回一个引用确实是在浪费时间，但是有些时候你会想："好吧，**一个**静态值不够，那么用一个静态**数组**总可以了吧…"

我无法用实例来捍卫我的观点，但是我可以用非常简明的推理证明这样做会让你多羞愧：首先，你必须确定一个 **n** 值，也就是数组的大小。如果 **n** 太小了，函数返回值的存储空间可能会用完，这种情况与刚才否定的单一静态对象的方案一样糟糕。

但是如果 **n** 的值太大，那么你的程序将面临性能问题，这是因为数组中的**每个**对象都应在函数在第一次调用时被构造。这会使你付出 **n** 次构造函数和 **n** 次析构函数[①]调用的代价，即使我们讨论的函数只被调用一次。如果将“优化”称为改善软件性能的一个步骤，那么我们可以把这一做法称为“劣化”。最后，请考虑一下：你如何将需要的值放入数组中的对象里，在放置的过程中你又付出了多大代价呢？在两个对象之间传值的最直接的方法就是赋值，但是赋值操作又会带来多大开销呢？对于许多类型而言，赋值的开销类似于调用一次析构函数（以销毁旧数值）加上一次构造函数（以复制新数值）。但是要知道，你的原始目标本来是避免构造和析构过程所带来的开销！请面对它：这样做一定不会得到好结果。（别妄想，用 `vector` 来代替数组也不会改善多少。）

在编写必须返回一个新对象的函数时，正确的方法就是让这个函数返回一个新对象。对于 `Rational` 的 `operator*`来说，这就意味着下面的代码是基本符合要求的：

```
inline const Rational operator*(const Rational& lhs, const Rational&
rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

显然地，这样做可能会招致对 `operator*`的返回值的构造和析构过程的开销，但是从长远角度讲，付出这小小的代价可以获得更大的收益。而且，这一可能会吓到你的清单也许永远不需要你来付账。就像其它编程语言一样，C++允许编译器的具体实现版本在不改变其固有行为的同时通过优化代码来提升性能，在某些情况下，对 `operator*`返回值的构造和析构过程可以被安全的排除。当编译器利用了这一事实（编译器通常都会这样做），你的程序就可以继续按预期的行为执行，同时还会比你预期的快一些。

归根结底，是使用引用返回，还是直接返回一个对象？你的工作就是：做出正确的抉择，使程序拥有正确的行为。然后把优化工作留给编译器制造商，他们会致力于让你的选择执行起来更高效。

[①] 程序结束时析构函数将调用一次。

时刻牢记

- 对于局部的/分配于栈上/分配于堆上的对象，如果你需要将其中的任意一种作为函数的返回值，请确保做到以下几点：不要返回一个指向局部的、分配于栈上的对象；不要返回一个引用去指向分配于堆上的对象；不要返回一个指向局部静态对象的指针或引用。（另有条目 4 中包含一个示例告诉我们：至少在在单线程环境下，返回一个指向局部静态对象的指针还是有意义的。）

条目22： 将数据成员声明为私有的

好吧，以下是我们的计划：我们首先要分析为什么数据成员不应该是公有的，然后继续分析为什么数据成员也不能是 `protected` 的。然后就引出本条款的结论：数据成员必须是私有的。结论引出，计划完成。

那么，数据成员为什么不能是 `public` 的？

让我们从讨论语义一致性问题开始（另请参见条目 18）。如果数据成员不是公有的，那么客户要想访问对象就只剩下成员函数一种方法。如果公有接口中所有的东西都是函数，那么客户在期望访问类成员时，由于一切都是函数，所以就可以任意使用，而不用担心是否需要使用括号。在整个过程中，这样做可以让你节省大量踌躇不定的时间。

但是也许你会发现，并没有强制规定来要求语义的一致性。那么你是否会发现：使用函数可以让你更精确地控制数据成员的访问权？如果把一个数据成员定义为 `public` 的，那么每个人对其都拥有“可读可写”的访问权，但是如果你使用函数来为数据成员赋值，或者获取数据成员的值，那么你可以将其实现为“禁止访问”、“只读”以及“可读可写”几种级别的访问权；嘿，如果需要，你甚至可以将其实现为“只写”的访问权：

```
class AccessLevels {
public:
  ...
  int getReadOnly() const          { return readOnly; }
  void setReadWrite(int value)     { readWrite = value; }
```

```
  int getReadWrite() const          { return readWrite; }
  void setWriteOnly(int value)      { writeOnly = value; }

private:
  int noAccess;                     // 此 int 值禁止访问
  int readOnly;                     // 此 int 值拥有只读级别访问权
  int readWrite;                    // 此 int 值拥有可读可写级别访问权
  int writeOnly;                    // 此 int 值拥有只写级别访问权
};
```

很有必要将访问权管理得如此有条不紊，因为许多数据成员**本应该**被隐藏起来。并不是每个数据成员都需要一个取值器（getter）和一个赋值器（setter）。

还不是十分肯定？那么现在是时候使出杀手锏了："封装"。如果你通过程序实现了对一个数据成员的访问，那么你就可以使用一次计算来代替这个数据成员，使用这一个类的人完全不会有所察觉。

请看下边的示例，假设你正在为一种自动装置编写一个应用程序，这一装置可以监视通过汽车的行驶速度，当一辆汽车通过时，这一应用程序就会计算出它的速度，然后将这一数值保存到一个小型数据库中，其中保存着曾通过所有车辆的速度数据：

```
class SpeedDataCollection {
  ...
public:
  void addValue(int speed);         // 添加新的数据值

  double averageSoFar() const;      // 返回速度的平均值

  ...
};
```

现在请注意成员函数 averageSoFar 的具体实现问题。一种实现方法是：为类添加一个数据成员，让它保存速度的平均值，随数据库的改动更新这一成员的数值。当调用 averageSoFar 时，它仅仅返回这一数据成员的值。另一种做法是：在每次调用 averageSoFar 时都计算出这一平均值，此时需要检查数据库中所有的数据值。

因为第一种手段（保存即时更新的平均值）中，你需要为保存即时更新平均值、累计总和以及数据的个数这几种数据成员分配空间，因此这一方法使得SpeedDataCollection对象都变得更大一些。然而，averageSoFar却十分的高效。可以把它写成一个内联函数（参见条目30），所做的仅仅是返回这一即时更新的。相反地，在需要时进行计算，averageSoFar速度上会慢一些，但是SpeedDataCollection对象的体积更小。

二者孰优孰劣，谁又能断定呢？在一个内存较为局促的机器（比如嵌入式的公路设备）上，并且该应用程序不会频繁的调用平均值，那么实时计算的方案就更为优秀。相反地，在平均值需要频繁使用，速度是程序的关键，内存不是问题的情况下，则更应采用保存一个即时平均值的方案。最重要的一点是，在通过成员函数访问平均值时（也就是"封装"），你可以交替使用这两种实现方案（当然，你可能还会想到其它重要的问题），客户顶多要做的一件事就是重新编译一下代码。（即使编译所带来的不方便也可以排除。参见条目31中介绍的技术。）

将数据成员隐藏在函数式接口的背后可以使得任意种类的实现方法更加灵活多变。比如说，这样做可以非常容易地做到下面几件事情：在数据成员进行读写操作时告知其他对象，验证类的恒定性和函数运行前后的状态，在多线程系统下进行同步操作，等等。如果让Delphi或C#的程序员使用C++，他们会发现C++这一特性与这些语言中的"属性"很相像，只是C++中需要添加一对括号。

封装是C++的一个博大精深的特性。如果你对客户隐藏了数据成员的话（也就是将它们封装起来），你就可以确保类永远保持一致性，这是因为只有成员函数可以影响到数据成员，同时你也保留了在以后改变具体实现方法的权利。如果你不将这些方法隐藏起来，那么你很快就会发现，即使你拥有类的源代码，对公有接口的修改也是受到严格限制的，因为这样做会破坏许多客户端代码。公有就意味着未封装，同时从实用角度讲，未封装就意味着无法更改，较为广泛应用的类更甚之。然而广泛应用的类最需要使用封装，因为它们可以从"具体实现可以不断改良"这一点上获得最大程度的收益。

上面的分析对于protected数据成员也适用。尽管二者乍看上去有一定的区别，但实际上它们是完全一致的。在使用public数据成员时，我们分析了语意一致性问题和访问权条理性问题，这一分析过程对于使用protected数据同样适用。但

还有一个问题——封装。protected 数据成员不是比 public 的更具有封装性吗？从实用角度讲，你会得到一个令人吃惊的答案：不是。

条目 23 中将介绍这一问题：C++中封装程度与代码的健壮程度（这段代码相关部分被修改时抵御破坏的能力）成正比。因此，数据成员的封装程度与代码的健壮程度也是成正比的。比如，当一个数据成员从类中移除时（可能你期望使用一次计算来代替，就像上文中的 averageSoFar 一样），代码是否会遭到破坏，将取决于封装程度。

请考虑这个问题：假设我们有一个 public 数据成员，然后我们把它删除了，那么将有多少的代码将遭到破坏呢？我们说，所有使用它的客户端代码。这将是一个**无法预知的巨大**数字。公有数据成员就是这样完全没有封装性的。但是继续考虑：我们有一个 protected 数据成员，然后我们把它删除了，此时将破坏多少代码？我们说，所有使用它的派生类，这同样是一个**无法预知的巨大**数字。由于在这两种情况下，如果数据成员被更改了，那么将会为客户带来无法估量的损失，因此可以说 protected 数据成员与 public 的一样没有封装性。这是违背直觉的，但是有经验的库实现者会告诉你，这是千真万确的。一旦你声明了一个 public 或 protected 的数据成员，然后客户开始使用它，你就很难再对这一数据成员做出修改。因为这样做会带来太多的代码重写、重新测试重新编写文档和重新编译等等工作。按封装的理念来说，对于数据成员仅仅存在两个层次的访问权，那就是：private（可以提供封装性）和非 private（不提供封装性）。

时刻牢记

- 要将数据成员声明为私有的。这样可以让客户端访问数据时拥有一致的语义，提供有序的访问控制，强制类保持一致性，为类作者提供更高的灵活性。
- protected 并不会带来比 public 更高的封装性。

条目23： 多用非成员非友元函数，少用成员函数

请假想一个表示网页浏览器的类。这个类可以提供诸多功能，其中包括清除下载缓存、清除访问历史、删除系统中保存的 cookie 等等：

```
class WebBrowser {
public:
  ...
  void clearCache();
  void clearHistory();
  void removeCookies();
  ...
};
```

许多用户可能需要同时执行这些操作，所以 WebBrowser 类应该提供一个函数来做这件事情：

```
class WebBrowser {
public:
  ...
  void clearEverything();          // 调用 clearCache、clearHistory
                                   // 以及 removeCookies
  ...
};
```

当然，这一功能也可以通过使用一个非成员函数调用适当的成员函数来实现：

```
void clearBrowser(WebBrowser& wb)
{
  wb.clearCache();
  wb.clearHistory();
  wb.removeCookies();
}
```

哪一个更好呢？是成员函数 clearEverything，还是非成员函数 clearBrowser？

面向对象的基本原理要求数据和对其进行操作的函数应该被包装在一起，同时建议成员函数为更优秀的选择。但不幸的是，这一建议并不是正确的。它是建立在对“面向对象的东西意味着什么”这一点的误解之上的。面向对象的基本原理要求数据应该尽可能的**被封装**起来。通过理性分析可以得知，成员函数 clearEverything 的封装性实际上比非成员函数 clearBrowser 还要**差**。还有，非成员函数可以为 WebBrowser 相关的功能提供更便利的打包方法，从而减少编译时依赖，提高 WebBrowser 的可扩展性。很多情况下，非成员函数的方法都比成员函数的方法要好。理解这一结论的原因是十分重要的。

我们从封装问题开始。如果一个物件被封装了，那么它就是不可见的。它的封装度越高，其它人或物件能看到它的机会就越少。能看到它的东西越少，我们对其进行修改的灵活性就越高，因为只有很少的物件能看到我们的改动。也就是说，一个物件的封装度越高，系统赋予我们修改它的能力就越强。这就是我们为什么将封装置于首要位置的原因：他为我们提供了改变物件的灵活性的方法，使用这一方法只会有很少的客户会受到影响。

请考虑与对象相关的数据。可以看到（也就是访问）某一数据的代码越少，就有更多的数据被封装起来，从而我们修改这一对象的数据特征（比如数据成员的个数、类型等等）时就更为自由，粗略计算一下某一数据可以被多少代码所访问，我们就可以计算出有多少函数可以访问这一数据：可以访问它的函数越多，这一数据的封装度就越低。

条目 22 中解释了数据成员为什么要声明为私有的，因为如果不这么做，那么就有无穷的函数可以访问它们。它们就毫无封装性可言。对于声明为私有的数据成员来说，可以访问它们的函数的个数就等于成员函数的个数加上友元函数的个数，因为只有成员和友元可以访问类中的私有数据。无论是成员函数（不仅仅可以访问类中的私有数据，还可以访问私有函数、enum 类型、由 typedef 生成的类型符号，等等）还是非成员非友元函数（上述成员函数可以访问的所有内容都不可访问）所提供的功能都是完全相同的，选择非成员非友元函数可以带来更完整的封装度，因为它不会使可以访问类中私有部分函数的数量增加。这就解释了为什么使用 clearBrowser（非成员非友元函数）比 clearEverything（成员函数）更理想：clearBrowser 可以为 WebBrowser 类提供更高的封装度。

此刻我们需要注意两件事情。第一，上面的推理过程仅仅适用于非成员**非友元**函数。由于友元对类中的私有成员的访问权与成员函数相仿，因此使用友元对封装度带来的影响与成员函数是一致的。如果充分考虑封装的因素，我们并不是在成员或非成员函数之间做出选择，而是成员函数和非成员非友元函数之间做出选择。（当然，封装也不是唯一需要考虑的因素。条目 24 中将为你介绍，当问题转向“隐式类型转换”时，选择就在成员和非成员函数之间进行。）

需要关注的第二件事仅仅是由于：封装要求函数不应为类一个成员，而并不意味着要求它也不是其它类的成员。在某些语言中（比如 Eiffel、Java、C#等等），所有函数**必须**包含在类中。对于习惯于使用这些语言的程序员来说，这第二件事多多少少可以算是对他们的一剂安抚药。比如说，我们可以将 clearBrowser 定义为某个“实用工具类”中的静态成员函数。只要它不是 WebBrowser 的一部分（或它的友元），它就不会影响到 WebBrowser 中私有成员的封装性。

在 C++中可以使用一个更为自然的方法：将 clearBrowser 定义为一个非成员函数，并将其与 WebBrowser 放置在同一个名字空间中：

```
namespace WebBrowserStuff {
  class WebBrowser { ... };

  void clearBrowser(WebBrowser& wb);

  ...
}
```

然而，你得到的东西远远要比更自然的代码要多。因为与类不一样的是，名字空间可以延伸至多个源代码文件中。这一点十分重要。ClearBrowser 这样的函数是“**便利函数**”。由于它们既不是成员函数也不是友元，所以它在访问 WebBrowser 时就没有任何特权，从而它也就不能够以其它的什么办法提供 WebBrowser 的客户端代码所不具备的功能。举例说，如果 clearBrowser 不存在的话，那么客户就只能自己动手调用 clearCache、clearHistory 和 removeCookies 这些函数了。

一个类似 WebBrowser 的类可能会有大量的便利函数，一些是关于书签的，另一些是关于打印的，还有关于 cookie 管理的，等等。作为一个一般守则，大多数客

户只会对这些便利函数中的一部分感兴趣。举例说，一个客户可能只对与书签相关的便利函数感兴趣，但是书签相关便利函数又依赖于 cookie 相关的便利函数，没有理由让这个程序员去关心那些额外信息。将着些便利函数分离开来的最直接的办法就是：将书签相关的便利函数声明在一个头文件中，cookie 相关的为与另一个头文件，打印相关的在第三个：

```
// 头文件 "webbrowser.h" – 为 WebBrowser 自身所定义的头文件
// 同时也包含“核心的” WebBrowser 相关的功能
namespace WebBrowserStuff {
   class WebBrowser { ... };
     ...                                  // “核心”相关功能
                                          // 比如几乎所有客户所需的非成员函数
}

// 头文件 "webbrowserbookmarks.h"
namespace WebBrowserStuff {
  ...                                     // 书签相关的便利函数
}

// 头文件 "webbrowsercookies.h"
namespace WebBrowserStuff {
  ...                                     // cookie 相关的便利函数
}

...
```

请注意：上面就是 C++标准库的组织方式。标准库使用了多个头文件（包括<vector>、<algorithm>、<memory>等等），每一个都声明了 std 名字空间中的**某一些**功能。而不是使用单一的、庞大的<C++StandardLibrary>头文件，并将 std 中所有的功能都罗列于此。对于仅希望使用 vector 相关功能的客户，不应该强迫他们去#include <memory>；不希望使用 list 的客户端也不需要去#include <list>。这样就使得所有客户仅仅需要考虑他们正在使用的那部分系统中的编译依赖问题。（参见条目 31，其中介绍了解决减缓编译依赖问题的其它途径。）让我们重新考虑成员函数的方案，以这种方式分开管理功能是不可行的，因为一个类必须要保证其完整性，它不能够被分割成块。

将所有的便利函数放置于多个头文件中（但位于同一个名字空间中），同时也意味着客户可以方便地**扩展**便利函数集。他们所需要做的仅仅是向同一名字空间中添加更多的非成员非友元函数。比如说，如果一个 WebBrowser 的客户希望编写一系列用于下载图片的便利函数，他或她仅仅需要在 WebBrowserStuff 名字空间中创建一个新的头文件来声明这些函数。新的函数与其它的便利函数一样可用，一样具有整合性。这是类无法提供的又一特性，因为类定义并不为客户提供扩展性。当然，客户可以派生新类，但是派生类仍无法访问基类中的封装（即私有）成员，所以我们说这样的"扩展功能"只有"二等"身份。同时，如同条目 7 中所讲，并不是所有的类都设计成了基类。

时刻牢记

- 多用非成员非友元函数，少用成员函数。这样做可以增强封装性，以及包装的灵活性和功能的扩展性。

条目24： 当函数所有的参数都需要进行类型转换时，要将其声明为非成员函数

我在这本书的序言中曾特别提到过，让类支持隐式类型转换在一般情况下都不会是一个好主意。当然，这一准则还是存在一些例外的，其中最普通的一个就是数值类型。举例说，如果你正在设计一个表示有理数的类，提供从整数向有理数的隐式转换也不是毫无道理的。很显然，这样做与 C++内建的从 int 向 double 的转换一样符合常理（甚至比 C++内建的从 double 向 int 的转换要符合常理得多）。这是千真万确的，你可能以这样的方式开始编写你的 Rational（有理数）类：

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
                                    // 构造函数是有意声明为非显性的，
                                    // 从而可以提供 int 到 Rational 的隐性转换

  int numerator() const;            // 用于访问分子和分母的函数
  int denominator() const;          // 参见条目 22
```

```
private:
  ...
};
```

此时你需要让这个类支持诸如加法、乘法等算术操作，但是你并不能确定这些操作是应该通过成员函数实现，还是以非成员函数的形式实现，亦或在可能的情况下以友元函数的形式实现。在你举棋不定的时候，你的本能会告诉你你应该尽量做到面向对象。你知道这一点，于是会说，有理数的乘法操作与 Rational 类相关，因此很自然地，有理数的 operator*就应该实现为 Rational 类内部的成员。与直觉恰恰相反的是，将函数放在相关的类中在有些时候恰恰是**违背**面向对象原则的（条目 23 中讨论过），我们暂时不考虑这一问题，考察一下用 operator*作为 Rational 的一个成员函数：

```
class Rational {
public:
 ...

 const Rational operator*(const Rational& rhs) const;
};
```

（如果你不太了解为什么以这种方式定义函数：返回一个 const 值，使用一个“const 引用”类型的参数，请参见条目 3、20、21）

这种设计方案会使乘法操作非常简便：

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth; // 工作正常

result = result * oneEighth;       //工作正常
```

但是你不能满足于现状。你可能期望 Rational 支持混合模式操作，也就是说 Rational 应该可以与其它类型值（比如 int）相乘。毕竟说，两数相乘的操作再自然不过了，即使这两个数的类型不一致。

然而，当你尝试进行混合模式算术时，你会发现它仅仅在一半的时间内正常工作：

```
result = oneHalf * 2;              // 工作正常

result = 2 * oneHalf;              // 出错！
```

这是一个不好的兆头。你是否记得乘法交换率呢？

如果你将上述后两个示例重写为它们等价的函数形式，代码中的问题就会浮出水面：

```
result = oneHalf.operator*(2);     // 工作正常

result = 2.operator*(oneHalf);     // 出错！
```

oneHalf 对象是一个类的实例，这个类中包含 operator*，于是编译器就会调用这个函数。然而整数 2 没有相关的类，因此就没有相关的 operator*成员函数。编译器仍然会去寻找 operator*的非成员函数版本（应该存在于名字空间域或者整体域），这些 operator*应该可以这样调用：

```
result = operator*(2, oneHalf);    // 出错！
```

但是在本示例中，没有任何非成员的 operator*能接收一个 int 和一个 Rational，因此搜寻工作自然会失败。

请再次关注一下调用成功的示例。你可以看到它的第二个参数是整数 2，而 Rational::operator*本身只将 Rational 作为它的参数类型。这里发生了什么呢？2 为什么仅在一种情况下正常运行，而另一种又不可以了呢？

这里发生的事情是：隐式类型转换。编译器知道你正在传入一个 int，而函数所需要的参数却是 Rational，但是编译器同时也知道它可以通过使用你所提供的 int 值作为参数，调用 Rational 的构造函数，从而“变出”一个合适的 Rational 来。也就是说，编译器在处理上述代码时，会以近似于下面的形式进行：

```
const Rational temp(2);            // 以 2 为参数，创建一个
                                   // 临时的 Rational 对象

result = oneHalf * temp;           // 与 oneHalf.operator*(temp)等价
```

当然，编译器这样做仅仅是因为有一个非显性的构造函数为其助一臂之力。如果 Rational 的构造函数是 explicit 的，那么下面的语句都是通不过编译的：

```
result = oneHalf * 2;              // 出错！（存在 explicit 的构造函数）
                                   // 无法将 2 转型为 Rational

result = 2 * oneHalf;              // 同样的错误，同样的问题
```

虽然上述的两条语句无法支持混合模式算术，但是至少二者的行为依旧保持一致。

然而你的目标是：即能保持一致性，又能支持混合模式算术，换句话说，寻找出让上述两条语句均能通过编译的设计方案。让我们返回先前的两条语句，来讨论一下：为什么即使 Rational 的构造函数不是 explicit 的，二者依然是一条可通过编译，另一条则通不过：

```
result = oneHalf * 2;              // 工作正常（oneHalf 拥有非显性构造函数）

result = 2 * oneHalf;              // 错误！（即使 oneHalf 拥有非显性构造函数）
```

看上去似乎**仅当这些参数存在于参数表中时**，它们才有资格进行隐式类型转换。而那些与调用成员函数的对象（也就是 this 所指向的对象）相关的隐式参数**永远也没有**资格进行隐式转换。这就是为什么第一次调用能够通过编译，而第二次不行。第一种情况涉及到参数表中所列的一个参数，而第二种没有。

但是此时你仍期望支持混合模式算术，同时此时工作方案也水落石出了：将 operator*声明为非成员函数，这样就可以允许编译器对所有参数进行隐式类型转换：

```
class Rational {
  ...                              // 不包含任何 operator*
};

const Rational operator*(const Rational& lhs, const Rational& rhs)
                                   // 将 operator*声明为非成员函数
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
```

```
}

Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;             // 工作正常

result = 2 * oneFourth;             // 太棒了！这样也可以了。
```

故事终于有了一个完美的结局，但是还留下了一处悬念。operator*是否应该做为 Rational 类的一个友元呢？

在这种情况下，答案是：不行。因为 operator*可以通过 Rational 的公用接口得到完整的实现。上面的代码交待了如何做这件事情。我们可以从中观察总结出一条重要结论，那就是：与成员函数相反的是**非成员函数**，而不是友元函数。有太多的 C++程序员自认为，如果一个函数与一个类相关，且不应将其实现为成员（比如说，所有参数都需要进行类型转换）时，应将其实现为友元。这个示例表明这样的推理是存在漏洞的。要尽量避免使用友元，因为，与现实生活中的情况类似，朋友为我们带来的麻烦往往要比好处多得多。当然不排除存在一些真挚的友谊。但是这并不意味着一个函数不应该作为成员时，就必须成为一个友元。

本条款中包含着真理，仅仅包含真理，但又不是真理的全部。当你从面向对象的 C++过渡至包含模板的 C++时（参见条目 1），你会将 Rational 实现为类**模板**而不是普通的类，此时就需要考虑新的问题了，也有了新的解决办法，一些设计实现的方法是不可思议的。这些问题、解决方案、具体实现是条目 46 讨论的主题。

时刻牢记

- 如果你需要对一个函数的所有参数全部进行类型转换（包括 this 指针所指的参数），那么它必须是一个非成员函数。

条目25： 最好不要让 swap 抛出异常

swap 是一个非常有趣的程序。它起初是作为 STL 的一部分引入 C++的，而后就成为了异常安全编程的一个重要的支柱（参见条目 29），同时对于可以自赋值的对

象而言它还是一个常用的复制处理机制。由于 swap 如此神通广大，那么以一个恰当的方式去实现它就显得十分重要了，但是它举足轻重的地位也决定了实现它并不是一件手到擒来的事情。在本条目中，我们就会针对 swap 函数展开探索，逐步掌握如何去驾驭它。

swap 函数的功能是**交换**两个对象的值。在默认情况下，交换工作是通过标准库的 swap 算法完成的。它的标准实现方式就能精确地完成你所期望的工作：

```
namespace std {

  template<typename T>          // std::swap 的标准实现
  void swap(T& a, T& b)         // 交换 a 与 b 的值
  {
    T temp(a);
    a = b;
    b = temp;
  }

}
```

只要你的类型支持复制（通过拷贝构造函数和拷贝复制运算符），那么默认的 swap 实现就可以让两个该类型的对象互相交换，你不需要专门做任何工作来支持这一功能。

然而，你可能对默认的 swap 实现抱有诸多不满。它会带来 3 次对象复制工作：a 复制到 temp，b 到 a，temp 到 b。对于一些类型来说，这些复制操作并不都是必需的。对于这些类型来说，默认的 swap 会成为你程序的桎梏。

上述的那种类型大都符合下面的特征：它的主要成分是一个指针，这一指针会指向另一个类型，真实的数据包含在另一个类型中。对这一设计的一种常见的形式是“pimpl 惯用法”（pointer to implementation，指向实现的指针，参见条目 31）。举例说明，Widget 类可以使用这种设计模式：

```
class WidgetImpl {                    // 保存 Widget 的数据的类
public:                               // 细节不重要
  ...
```

```
private:
  int a, b, c;                                  // 可能会有很多数据
  std::vector<double> v;                        // 复制它们的代价是很高的！
  ...
};

class Widget {                                  // 使用 pimpl 惯用法的类
public:
  Widget(const Widget& rhs);

  Widget& operator=(const Widget& rhs)          // 要复制一个 Widget 对象，只要
  {                                             // 复制对应的 WidgetImpl 对象。
   ...                                          // 关于 operator=实现的一般信息
   *pImpl = *(rhs.pImpl);                       // 参见条目 10、11、12
   ...
  }

  ...

private:
  WidgetImpl *pImpl;                            // 指向包含当前 Widget 数据的对象
};
```

为了交换两个 Widget 对象的值，我们所要做的仅仅是交换他们的 pImpl 指针，但是默认的 swap 算法是不可能知道这一切的，它不仅会复制三个 Widget 对象，同时也会复制三个 Widget 对象。这样做效率太低了。

我们要做的是告诉 std::swap 当交换 Widget 时，执行的交换操作应当仅仅针对它们内部的 pImpl 指针。有一种精确的说法来描述这一方法：将 Widget 的 std::swap 特化。下面是基本的思想，尽管以这种方式不能通过编译：

```
namespace std {

  template<>                            // 在 T 为 Widget 时，
  void swap<Widget>(Widget& a,          // 这是 std::swap 的一个特化版本
```

```
                  Widget& b)        // 这段代码不能通过编译
  {
    swap(a.pImpl, b.pImpl);         // 要交换两个 Widget，
  }                                 // 只需要交换它们的 pImpl 指针

}
```

程序开端的“template<>”告诉我们这是 std::swap 的一个**完全特化模板**，函数名后面的“<Widget>”告诉我们当前的特化针对 T 是 Widget 的情况。换种说法，当一般的 swap 模板应用于 Widget 时，应当使用这一具体实现。一般情况下，我们没有权限去改动 std 名字空间内部的内容，但是我们有权针对我们自己创建的类型（比如 Widget）来完整地特化标准模板（就像 swap）。这就是我们所要做的。

然而，就像我说过的，这段代码是不能通过编译的。这是因为它尝试访问 a 与 b 内部的 pImpl 指针，但是它们是私有的。我们可以将我们的特化函数声明为友元，但这里的规则有些不同：这里要求我们让 Widget 包含一个名为 swap 的公共成员函数，让这个 swap 进行实际的交换工作，然后特化 std:swap 来调用这一成员函数。

```
class Widget {                      // 同上，
public:                             // 仅添加了一个 swap 成员函数
  ...
  void swap(Widget& other)
  {
    using std::swap;                // 本节后面会解释为什么这样声明

    swap(pImpl, other.pImpl);       // 交换 pImpl 指针来交换 Widget
  }
  ...
};

namespace std {

  template<>                        // 特化的 std::swap （已修正）
  void swap<Widget>(Widget& a, Widget& b)
```

```
  {
    a.swap(b);                          // 要交换 Widget,
  }                                     // 只要调用它们的 swap 成员函数

}
```

这样的代码不仅仅可以通过编译，而且也与 STL 容器相协调，它不仅仅提供了公有的 swap 成员函数，而且还提供了特化的 std::swap 来调用这些成员函数。

然而，我们不难发现，Widget 和 WidgetImpl 都是类**模板**，而不是类，似乎我们可以自定义 WidgetImpl 中保存的数据的类型：

```
template<typename T> class WidgetImpl { ... };

template<typename T> class Widget { ... };
```

将一个 swap 成员函数放入 Widget 中（如果需要，也可以是 WidgetImpl）仍然十分简单，但是我们对 std::swap 特化时将会遇到问题。下面是我们希望编写的代码：

```
namespace std {
  template<typename T>
  void swap<Widget<T> >(Widget<T>& a, Widget<T>& b)
                                        // 错误！非法代码
  { a.swap(b); }

}
```

这样的代码看上去完美无瑕，但是它是非法的。因为其中尝试对一个函数模板（std::swap）进行不完全的特化，但是，尽管 C++允许对类模板进行不完全特化，而函数模板就不行了。这一代码不应通过编译(尽管一些编译器会错误的接受)。

当你期望对一个函数模板进行"不完全特化"时，通常的做法非常简单，就是添加一个该函数的重载。代码可能是下面的样子：

```
namespace std {

  template<typename T>                  // std::swap 的一个重载
```

```
  void swap(Widget<T>& a, Widget<T>& b)
                                    // （注意 swap 后边没有<...>）
  { a.swap(b); }                    // 下文解释了为什么这样做不合法

}
```

一般情况下，重载函数模板是可以的，但是 std 是一个很特殊的名字空间，它的规则也是独特的。对 std 中的模板进行完全特化是合法的，但是为 std 添加一个**新的**模板却是不合法的（类或函数或其他一切都不可以）。std 的内容是由 C++标准化委员会一手确定的，我们无法修改他们所规定的任何形式，只能“望码兴叹”。越轨的代码似乎可以运行，但它们的行为却是未定义的。如果你希望你的代码拥有可预知的行为，你就不应该在 std 中添加新的内容。

那么应该怎么办呢？我们仍然需要一种方法来让其他人通过调用 swap 来访问我们更加高效的特化版本。答案很简单。我们仍然可以通过声明一个非成员函数 swap 来调用成员函数 swap 实现，只要这个非成员函数不是 std::swap 的特化或者重载版本即可。比如说，如果我们所有与 Widget 相关的功能都在名字空间 WidgetStuff 中，那么代码看上去应该是这样：

```
namespace WidgetStuff {
  ...                               // 模板化的 WidgetImpl，等等

  template<typename T>              // 同上，包括 swap 成员函数
  class Widget { ... };
  ...

  template<typename T>              // 非成员函数 swap
  void swap(Widget<T>& a, Widget<T>& b)
                                    // 不属于 std 名字空间
  {
    a.swap(b);
  }
}
```

现在，如果任意位置的代码对两个 Widget 对象调用了 swap，C++的名字搜寻守则（更具体地说，就是所谓的**参数依赖搜寻**或 **Koenig 搜寻**）将会在 WidgetStuff 中查找具体到 Widget 的版本。这恰恰是我们需要的。

由于这种方法针对类或者类模板可以正常运行，所以看上去似乎我们应该在任何情况下都使用它。但是遗憾的是，我们还是要对于类的 std::swap 进行特化（稍后会交代理由），所以如果你想要在尽可能多的上下文中（你所需要的）调用具体到类的 swap 版本，你就需要在你的类所在的名字空间编写一个非成员版本的 swap，同时还需要一个 std::swap 的特化版本。

顺便说一下，即使你没有使用名字空间，上述内容仍然有效（也就是说，你仍需要一个非成员的 swap 去调用成员函数 swap），但是为什么你要把所有的类、模板、函数、枚举类型、enumerant、typedef 的名字统统塞进全局名字空间里呢？如果你对编程规范有一点概念的话，都不会这样做的。

到目前为止我所介绍的一切内容都是以 swap 的作者的角度展开的，但是以一个客户的眼光来审视一下 swap 也是很有价值的。假设你正在编写一个函数模板，这里你需要交换两个对象的值：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  ...
  swap(obj1, obj2);
  ...
}
```

这里应该调用哪一个 swap 呢？ std 中存在一个通用版本，这是你所知道的；另外 std 中可能还有一个针对这一通用版本的特化版本，它可能存在也可能不存在；或者一个模板的版本，它可能存在也可能不存在，它是否在一个名字空间中也不能确定（但可以肯定不在 std 名字空间中）？此时你所希望的是，如果存在一个模板版本的话，就调用它；如果不存在，就返回调用 std 中的通用版本。以下是满足这一要求的代码：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
```

```
{
  using std::swap;                    // 确保 std::swap 在此函数中可用
  ...
  swap(obj1, obj2);                   // 为类型 T 的对象调用最佳的 swap
  ...
}
```

当编译器看到对 swap 的调用时，它们会寻找恰当的 swap 来进行调用。C++的名字搜寻原则确保了在全局或 T 类型所在的名字空间中来查找所有的精确到 T 的 swap。（举例说，如果 T 是位于 WidgetStuff 名字空间中的 Widget，那么编译器将会使用参数依赖搜寻方式来查找 WidgetStuff 中的 swap。）如果没有精确到 T 的 swap 存在，那么编译器将会使用 std 中的 swap，多亏了 using 声明可以使 std::swap 在本函数中可见。然而即使这样，编译器也更期望得到一个精确到 T 的 std::swap 的特化版本，而不是未确定类型的模板，因此如果 std::swap 特化为 T 版本，那么这一特化的版本将会得到使用。

因此，调用正确的 swap 十分简单。你所需要关心的事仅仅是不去限制对它的调用，因为如果这样做会使 C++如何决定去调用函数的方式受到影响。举例说，如果你用下面的方式调用了 swap：

```
std::swap(obj1, obj2);                // 调用 swap 的错误方法
```

你强迫编译器仅仅去考虑 std 中的 swap（包括所有的模板特化版本），这样做就排除了得到一个位于其他位置的精确到 T 版本的 swap 的可能，即使它是更加合理的。然而，一些进入误区的程序员还是会以这种方式限制 swap 的调用，这里你就可以看出，为你的类提供一个 std::swap 的完全特化版本是多么重要：对于那些使用不恰当的编码风格写出的代码（这样的代码也存在于一些标准库的实现当中，如果你感兴趣可以自己编写一些代码，来帮助这样的代码尽可能的提高效率），精确到类的 swap 实现仍然有效。

此刻，我们已经介绍了默认的 swap、成员 swap、非成员 swap、std::swap 的特化版本，以及对 swap 的调用，现在让我们来做一个总结。

首先，如果对你的类或者类模板使用默认的 swap 实现能够得到可以接受的效率，你就不需要做任何事情。任何人想要交换你创建的类型的对象时，都会去调用默认的版本，此时可以正常工作。

其次，如果默认的 swap 实现并不够高效（大多数情况下意味着你的类或模板正在运用 pimpl 惯用法），请按下面步骤进行：

1．提供一个公用的 swap 成员函数，让它可以高效的交换你的类型的两个对象的值。理由将在后面列出，这个函数永远不要抛出异常。
2．在你的类或模板的同一个名字空间中提供一个非成员的 swap。让它调用你的 swap 成员函数。
3．如果你正在编写一个类（而不是类模板），要为你的类提供一个 std::swap 的特化版本。同样让它调用你的 swap 成员函数。

最后，如果你正在调用 swap，要确保使用一条 using 声明来使 std::swap 对你的函数可见，然后在调用 swap 时，不要做出任何名字空间的限制。

文中还有一处欠缺，那就是本文的标题中的敬告：不要让 swap 的成员函数版本抛出异常。这是因为 swap 最重要的用途之一就是帮助类（或类模板）来提供异常安全的保证。条目 29 中详细介绍了这一点，但是这一技术做出了“swap 的成员函数版本永远不会抛出异常”这一假设。这一约束仅仅应用于成员函数版本，非成员版本则不受这一限制。这是因为 swap 的默认版本基于拷贝构造和拷贝赋值，而在一般情况下，这两种函数都可能抛出异常。因此，当你编写一个自定义版本的 swap 时，在典型情况下你不仅要提供一条更高效的交换对象值的方式，同时你也要提供一个不抛出异常的版本。作为一条一般的守则，这两条 swap 的特征是相辅相成的，因为高效的 swap 同时也基于内建数据类型的操作（诸如 pimpl 惯用法中使用的指针），同时内建数据类型的操作决不会抛出异常。

时刻牢记

- 在对你的类型使用 std::swap 时可能会造成效率低下时，可以提供一个 swap 成员函数。确保你的 swap 不要抛出异常。

- 如果你提供了一个 swap 的成员函数，那么同时要提供一个非成员函数 swap 来调用这一成员。对于类而言（而不是模板），还要提供一个 std::swap 的特化版本来调用 swap 成员函数。
- 在调用 swap 时，要为 std::swap 使用一条 using 声明，然后在调用 swap 时，不要做出名字空间的限制。
- 对用户自定义类型而言，提供 std 的完全特化版本不成问题，但是决不要尝试在 std 中添加全新的内容。

第五章. 实现

在大多数情况下，恰当地做好类（以及类模板）的定义和函数（以及函数模板）的声明是整个实现工作的重中之重。一旦你顺利地完成了这些工作，那么相关的实现工作大都是直截了当的。然而，这里还存在一些需要关注的事情：过早地定义变量可能会拖慢性能。滥用转型可能会使代码变得笨重且不易维护，同时也会困扰于无尽难于发现的 bug。返回一个对象内部内容的句柄会破坏代码的封装性，同时留给客户一个悬而未决的“悬空句柄”。如果对异常的影响考虑不周，那么将会带来资源泄露和数据结构的破坏。过分热衷于使用内联会使代码不断膨胀。代码文件过多过复杂会造成的过于复杂的耦合，程序的构建时间会漫长得让人无法忍受。

所有这些问题都是可以避免的。本章就来讲解如何去做。

条目26： 定义变量的时机越晚越好

你经常要使用构造函数或者析构函数来定义某个类型的一个变量，当系统在运行至变量的定义时，就会引入一次构造的开销；在变量达到自身作用域的边界时，就会引入一次析构的开销。未使用的变量也会带来一定的开销，所以你应该尽可能的避免这种浪费的出现。

你可能会想你永远也不会定义变量而不去使用，但是你可能需要三思而后行。请观察下边的函数，它在所提供的密码足够长时，可以返回一个加密版本的密码。如果密码长度过短，函数就会抛出一个 `logic_error` 类型的异常（这个异常类型定义于标准 C++库中，参见条目 54）：

```
// 这个函数定义"encrypted"变量的时机过早
std::string encryptPassword(const std::string& password)
{
  using namespace std;

  string encrypted;

  if (password.length() < MinimumPasswordLength) {
    throw logic_error("Password is too short");
```

```
  }
  ...                                   // 对密码加密
  return encrypted;
}
```

本函数中，尽管对象 encrypted 并不是完全未使用的，但是在抛出异常的情况下，函数就不会使用它。也就是说，即使 encryptPassword 抛出一个异常，你也要为 encrypted 付出一次构造和一次析构的代价。因此，你最好推迟 encrypted 的定义，直到你**确认**你需要它时再进行：

```
// 这个函数推迟了 encrypted 的定义，直到真正需要它时再进行
std::string encryptPassword(const std::string& password)
{
  using namespace std;

  if (password.length() < MinimumPasswordLength) {
     throw logic_error("Password is too short");
  }

  string encrypted;

  ...                                   // 对密码加密
  return encrypted;
}
```

上面的代码还没有那么严谨，这是因为在定义 encrypted 时没有为它设置任何初始化参数。这就意味着编译器将调用它的默认构造函数。通常情况下，你要对一个对象需要做的第一件事就是为它赋一个值，通常是通过一次赋值操作。条目 4 中解释了为什么使用默认构造函数构造对象并为其赋值，要比使用需要的值对其进行初始化的效率低一些。那里的分析符合此处的情况。比如说，可以假设的较困难的部分是通过下面的函数来解决的：

```
void encrypt(std::string& s);       // 适时为 s 加密
```

encryptPassword 就应该以下面的方式来实现了，尽管它不是最优秀的：

```
// 这一函数推迟了 enctypted 定义的时机，直到需要时才进行。
```

```
// 但仍然会带来不必要的效率问题。
std::string encryptPassword(const std::string& password)
{
  ...                                  // 同上，检查密码长度

  std::string encrypted;               // encrypted 的默认构造函数版本
  encrypted = password;                // 对 encrypted 赋值

  encrypt(encrypted);
  return encrypted;
}
```

更好的一种实现方式是，使用 password 来初始化 encrypted，这样就可以跳过默认构造过程所带来的无谓的性能开销：

```
// 最后给出定义和初始化 encrypted 的最佳方法
std::string encryptPassword(const std::string& password)
{
  ...                                  // 检查长度

  std::string encrypted(password); // 通过拷贝构造函数定义和初始化

  encrypt(encrypted);
  return encrypted;
}
```

此时标题中的“越晚越好”的真正含义就十分明显了。你不仅仅要推迟一个变量的定义时机，直到需要它时再进行；你还需要继续推迟，直至你掌握了它的初始化参数为止。这样做，你就可以避免去构造和析构不必要的对象，你也可以避免那些无关紧要的默认构造过程。还有，通过初始化这些变量，定义这些变量的目的一目了然，从而代码也变得更加清晰。

“但是循环呢？”你可能会想。如果一个变量仅仅在循环题中使用，那么更好的选择是：将它定义在循环题的外部，在每次循环迭代前对其进行赋值；还是：在循环体的内部定义变量？也就是说，哪种基本结构是更优秀的呢？

```
// 方法 A：在循环体外部定义
```

```
Widget w;
for (int i = 0; i < n; ++i){
  w = 取决于 i 的某个值;
  ...
}

// 方法 B: 在循环体内部定义

for (int i = 0; i < n; ++i) {
Widget w(取决于 i 的某个值);
  ...
}
```

这里我使用了 Widget 类型的对象，而不是 string 类型的对象，从而避免了进行构造、析构、或者对象赋值等过程带来的误差。

对于 Widget 的操作而言，上面两种方法所带来的开销如下：

- 方法 A：1 个构造函数 + 1 个析构函数 + n 次赋值。
- 方法 B：n 个构造函数 + n 个析构函数。

对于那些一次赋值操作比一对构造-析构操作开销更低的类而言，方法 A 是较高效的。尤其是在 n 较大的情况下。否则方法 B 就是更好的选择。还有，方法 A 使得 w 位于一个比方法 B 更大的作用域中，这是违背程序的可读性和可维护性原则的。因此，除非你确认: (1)赋值操作比一对构造-析构操作更高效，(2)当前代码是对性能敏感的；其他任何情况下，你都应该使用方法 B。

时刻牢记

- 定义变量的时机越晚越好。这可以提高程序的清晰度和工作效率。

条目27： 尽量少用转型操作

C++的设计初衷之一就是：确保代码远离类型错误。从理论上来讲，如果你的程序顺利通过了编译，那么它就不会对任何对象尝试去做任何不安全或无意义的操作。这是一项非常有价值的保证。你不应该轻易放弃它。

然而遗憾的是，转型扰乱了原本井然有序的类型系统。它可以带来无穷无尽的问题，一些是显而易见的，但另一些则是极难察觉的。如果你是一名从 C、Java 或者 C# 转向 C++的程序员的话，那么请注意了，因为相对 C++而言，转型在这些语言中更加重要，而且带来的危险也小得多。但是 C++不是 C，也不是 Java、C#。在 C++中，转型是需要你格外注意的议题。

让我们从复习转型的语法开始，因为实现转型有三种不同但是等价的方式。C 风格的转型是这样的：

```
(T) 表达式                    // 将表达式转型为 T 类型的
```

函数风格转型的语法如下：

```
T(表达式)                     //将表达式转型为 T 类型的
```

两者之间在含义上没有任何的区别。这仅仅是你把括号放在哪儿的问题。我把这两种形式称为“怀旧风格的转型”。

C++还提供了四种新的转型的形式（通常称为“现代风格”或“C++风格”的转型）：

```
const_cast<T>(表达式)
dynamic_cast<T>(表达式)
reinterpret_cast<T>(表达式)
static_cast<T>(表达式)
```

四者各司其职：

- const_cast 通常用来脱去对象的恒定性。C++风格转型中只有它能做到这一点。
- dynamic_cast 主要用于进行“安全的向下转型”，也就是说，它可以决定一个对象的类型是否属于某一个特定的类型继承层次结构中。它是唯一一种怀旧风

格语法所无法替代的转型。它也是唯一一种可能会带来显著运行时开销的转型。（稍候会具体讲解。）

- reinterpret_cast 是为底层转型而特别设置的，这类转型可能会依赖于实现方式，比如说，将一个指针转型为一个 int 值。除了底层代码以外，要格外注意避免这类转型。此类转型在这本书中我只用过一次，而也是在讨论如何编写一个针对未分配内存的调试分配器（参见条目 50）用到。

- static_cast 可以用于强制隐式转换（比如说，将一个非 const 的对象转换为 const 对象（就像条目 3 中一样），int 转换为 double，等等）。它可以用于大多数这类转换的逆操作（比如说，void*指针转换为包含类型的指针，指向基类的指针转换为指向继承类的指针），但是它不能进行从 const 到非 const 对象的转型。（只有 const_cast 可以。）

怀旧风格的转型在 C++中仍然是合法的，但是这里更推荐使用新形式。首先，它们在代码中更加易于辨认（不仅对人，而且对 grep 这样的工具也是如此），对于那些类型系统乱成一团的代码，这样做可以减少我们为类型头疼的时间。其次，对每次转型的目的更加细化，使得编译器主动诊断用法错误成为可能。比如说，如果你尝试通过转型脱去恒定性的话，你只能使用 const_cast，如果你尝试使用其它现代风格的转型，你的代码就不会通过编译。

需要使用怀旧风格转型的唯一的一个地方就是：调用一个 explicit 的构造函数来为一个函数传递一个对象。比如：

```
class Widget {
public:
  explicit Widget(int size);
  ...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                 // 函数风格转型
                                        // 基于 int 创建一个 Widget
```

```
doSomeWork(static_cast<Widget>(15));   // C++风格转型
                                       // 基于 int 创建一个 Widget
```

出于某些原因，手动创建一个对象“感觉上”并不像一次转型，所以在这种情况下应更趋向于使用函数风格的转型而不是 static_cast。同时，由于即使在你写下的代码可能会导致核心转储时，你仍然会“感觉”你有充足的理由那样做，因此你可能要忽略你的直觉，自始至终使用现代风格的转型。

许多程序员相信转型只是告诉编译器将一个类型作为另一种来对待，仅此而已，殊不知任何种类的类型转换（无论是显式的还是通过编译器隐式进行的）通常会在运行时引入一些新的需要执行的代码。比如说，在下面的代码片断中：

```
int x, y;
...
double d = static_cast<double>(x)/y;    // x 除以 y，用浮点数保存商值
```

int x 向 double 的转型几乎一定要引入新的代码，因为在绝大多数架构中，int 与 double 的底层表示模式是不同的。这并不那么令人吃惊，但是下面的示例也许会让你的眼界更加开阔些：

```
class Base { ... };

class Derived: public Base { ... };

Derived d;

Base *pb = &d;                          // 隐式转换： Derived* => base*
```

这里我们创建了一个基类的指针，并让其指向了一个派生类的对象，但是某些时候，这两个指针值并不会保持一致。如果真的这样了，系统会**在运行时**为 Derived*指针应用一个偏移值来取得正确的 Base*指针的值。

上文的示例告诉我们：一个单独的对象（比如一个 Derived 的对象）可能会拥有一个以上的地址（比如，一个 Base*指针指向它的地址和一个 Derived*指针指向它的地址）。这件事在 C 语言中是绝不会发生的。同样在 Java 或 C#中均不会发生。但在 C++中**的的确确**发生了。实际上，在使用多重继承时这件事几乎是必然的，而在单继承环境下也有可能发生。这意味着你应该避免去假设或推定 C++放置

对象的方式，同时你应该避免基于这样的假设来进行转型。比如，如果你将对象地址转型为 char*指针，然后再对其进行指针运算，通常都会使程序陷入未定义行为。

但是请注意，我说过“某些时候”才需要引入偏移值。对象放置的方法、地址计算的方法都是因编译器而异的。这就意味着，仅仅由于你“知道对象如何放置”，你对在某一个平台上转型的做法可能充满信心，但它在另一些平台上却是一钱不值。世界上有许多程序员为此付出了惨痛的代价。

关于转型的一件有趣事情是 :你很容易编写一些 “看上去正确”的东西，但实际上它们是错误的。举例说，许多应用程序框架需要在派生类中实现一个虚拟成员函数，并首先让这些函数去调用基类中对应的函数。假设我们有一个 Window 基类和一个 SpecialWindow 派生类，两者都定义了虚函数 onResize。继续假设：SpecialWindow 的 onResize 首先会调用 Window 的 onResize。以下是实现方法，它乍看上去是正确的，其实不然：

```
class Window {                              // 基类
public:
  virtual void onResize() { ... }           // 基类 onResize 的实现
  ...
};

class SpecialWindow: public Window {        // 派生类
public:
  virtual void onResize() {                 // 派生类 onResize 的实现
    static_cast<Window>(*this).onResize();
                                            // 将*this 转型为 Window,
                                            // 然后调用它的 onResize,
                                            // 这样不会正常工作！

    ...                                     // 完成 SpecialWindow 独有的任务
  }
  ...
};
```

上面代码中的转型操作已经用黑体字标出。（这是一个现代风格的转型，但是如果使用怀旧风格也不会带来任何改善。）就像你所预料的那样，代码将会将*this 转型为一个 Window，因此这里的 onResize 的调用应该是 Window::onResize。而你一定不会预料到，当前对象并没有调用这一函数。取而代之的是，转型过程创建了一个新的，*this 中基类部分的一个临时**副本**，然后调用这一副本的 onResize。上面的代码将不会调用当前对象的 Window::onResize，然后进行对象中的具体到 SpecialWindow 的动作；而是再对当前对象进行 SpecialWindow 行为之前，去调用当前对象的**基类部分的副本**中的 Window::onResize。如果 Window::onResize 希望修改当前对象（这也不是完全不可能，因为 onResize 是一个非 const 的成员函数），实际上当前对象不会受到任何影响。取而代之的是，这一对象的那个副本将会被修改。然而，如果 SpecialWindow::onResize 希望修改当前对象，当前对象**将会**被修改，这将导致下面的情景：代码将会使当前对象处于病态中——它基类部分的修改没有进行，而派生类部分的修改却完成了。

解决方案就是：避免转型。用你真正需要的操作加以替换。你并不希望欺骗编译器将一个*this 误识别为一个基类对象；你希望对当前对象调用 onResize 的基类版本。那么就这样编写好了：

```
class SpecialWindow: public Window {
public:
  virtual void onResize() {
    Window::onResize();             // 对*this 调用 Window::onResize
    ...
  }
  ...

};
```

这个示例同时告诉我们：如果你发现你的工作需要进行转型，那么此处就告诉了你当前的努力方向可能是错误的。尤其是在你期望使用 dynamic_cast 时。

在深入探究 dynamic_cast 的实现设计方式之前，有必要先了解一下：大多数 dynamic_cast 实现的运行速度是非常缓慢的。比如说，至少有一种普遍的实现

是通过比较各个类名的字符串。如果你正在针对一个四层深的单一继承层次结构中的一个对象进行 dynamic_cast，那么这种实现方式下，每一次 dynamic_cast 都会占用四次调用 strcmp 的时间用于比较类名。显然地，更深的或者使用多重继承的层次结构的开销将会更为显著。一些实现以这种方式运行也是有它的根据的（它们这样做是为了支持动态链接）。在对性能要求较高的代码中，要在整体上时刻对转型持谨慎的态度，你应该特别谨慎地使用 dynamic_cast。

一般说来，如果你确信一些对象是派生类的，那么当你对这些对象进行派生类的操作时，你只有一个指针或者一个指向基类的引用能操作这类对象，dynamic_cast 将可能派上用场。一般有两条途径来避免这一问题。

首先，可以使用容器来保存直接指向派生类对象的指针（通常是智能指针，参见条目 13），这样就无需通过基类接口来操作这些对象。比如说，在我们的 Window/SpecialWindow 层次结构中，如果只有 SpecialWindow 支持闪烁效果，我们也许可以这样做：

```
class Window { ... };

class SpecialWindow: public Window {
public:
  void blink();
  ...
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
// 关于 tr1::shared_ptr 的信息请参见条目 13

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin();  // 不好的代码。
     iter != winPtrs.end();                 // 使用 dynamic_cast
     ++iter) {
  if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get()))
```

```
    psw->blink();
}
```

这里有更好的解决方案：

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;

VPSW winPtrs;

...

for (VPSW::iterator iter = winPtrs.begin();  // 更好的代码
     iter != winPtrs.end();                  // 无需 dynamic_cast
     ++iter)
  (*iter)->blink();
```

当然，在使用这一方案时，在同一容器中放置的 Window 派生对象的类型是受到限制的。为了使用更多的 Window 类型，你可能需要多个类型安全的容器。

一个可行的替代方案是：在基类中提供虚函数，然后按需配置。这样对于所有可能的 Window 派生类型，你都可以通过基类接口来进行操作了。比如说，尽管只有 SpecialWindow 可以闪烁，但是在基类中声明这一函数也是有意义的，可以提供一个默认的实现，但不去做任何事情：

```
class Window {
public:
  virtual void blink() {}          // 默认实现不做任何事情；
  ...                              // 条目 34 将介绍：
};                                 // 提供默认实现可能是个坏主意

class SpecialWindow: public Window {
public:
  virtual void blink() { ... };    // 在这一类型中
  ...                              // blink 函数会做一些事情
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
```

```
VPW winPtrs;                          // 容器中保存着（指向）
...                                   // 所有可能的 Window 类型

for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end();
++iter)
                                      // 请注意这里没有 dynamic_cast
  (*iter)->blink();
```

上面的这两种实现（使用类型安全的容器，或者在层次结构的顶端添加虚函数）都不是万能的。但在大多数情况下，它们是 dynamic_cast 良好的替代方案。如果你发现其中一种方案可行，大可以欣然接受。

关于 dynamic_cast 有一件事情自始至终都要注意，那就是：避免级联式使用。就是说要避免类似下面的代码出现：

```
class Window { ... };
...                                   // 此处定义派生类
typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end();
++iter)
{
  if (SpecialWindow1 *psw1 =
      dynamic_cast<SpecialWindow1*>(iter->get())) { ... }
  else if (SpecialWindow2 *psw2 =
      dynamic_cast<SpecialWindow2*>(iter->get())) { ... }
  else if (SpecialWindow3 *psw3 =
      dynamic_cast<SpecialWindow3*>(iter->get())) { ... }
  ...
}
```

这样的代码经编译后得到的可执行代码将是冗长而性能低下的，并且十分脆弱，这是因为每当 Window 的类层次结构有所改变时，就需要检查所有这样的代码，以断定它们是否需要更新。（比如说，如果添加了一个新的派生类，上面的级联操作中

就需要添加一个新的条件判断分支。）这样的代码还是由虚函数调用的方式取代为好。

优秀的 C++代码中使用转型应该是十分谨慎的，但是一般说来并不是要全盘否定。比如说，前文[1]从 int 向 double 的转型，就是一次合理而有用的转型，尽管它并不是必需的。（代码可以这样重写：声明一个新的 double 类型的变量，并且用 x 的值对其进行初始化。）和其他绝大多数可以结构一样，转型应该尽可能的与其它代码隔离，典型的方法是将其隐藏在函数中，这些函数的的接口就可以防止调用者接触其内部复杂的操作。

时刻牢记

- 尽可能避免使用转型，尤其是在对性能敏感的代码中不要使用动态转型 dynamic_cast。如果一个设计方案需要使用转型，要尝试寻求一条不需要转型的方案来取代。
- 在必须使用转型时，要尝试将其隐藏在一个函数中。这样客户就可以调用这些函数，而不是在他们自己的代码中使用转型。
- 要多用 C++风格的转型，少用怀旧风格的转型。现代的转型更易读，而且功能更为具体化。

条目28： 不要返回指向对象内部部件的“句柄”

假设你当前设计的应用程序里会涉及到矩形。每个矩形的区域都由它的左上角和右下角的坐标来表示。为了让 Rectangle 对象尽可能的小巧，你可能会做出这样的决定：并不在 Rectangle 内部保存这些点的坐标信息，而是将这些信息保存在一个辅助结构中，然后让 Rectangle 指向它：

```
class Point {                          // 表示点的类
public:
```

[1] 141 页的 doSomeWork(Widget(15))，使用函数风格转型创建一个 int 的 Widget。——译注

```
  Point(int x, int y);
  ...

  void setX(int newVal);
  void setY(int newVal);
  ...
};

struct RectData {                    // 供 Rectangle 类使用的点的数据
  Point ulhc;                        // ulhc = "左上角"
  Point lrhc;                        // lrhc = "右下角"
};

class Rectangle {
  ...

private:
  std::tr1::shared_ptr<RectData> pData;
                                     // 关于 tr1::shared_ptr 请参见条目 13
};
```

因为 Rectangle 的客户可能需要计算矩形的面积，所以这个类就应该提供 upperLeft 和 lowerRight 函数。然而，Point 却是一个用户自定义的类型，因此伴着你对条目 20 中相关推论的回忆——通过引用传递用户自定义类型的对象要比直接传值更高效，你会让这些函数返回指向 Point 的引用：

```
class Rectangle {
public:
  ...
  Point& upperLeft() const { return pData->ulhc; }
  Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

这样的设计可以通过编译，但是它却是错误的。实际上，它是自我矛盾的。一方面，由于 upperLeft 和 lowerRight 的设计初衷仅仅是为客户提供一个途径来了解

Rectangle 的两个顶点坐标在哪里，而不是让客户去修改它，因此这两个函数应声明为 const 成员函数。另一方面，这两个函数都返回指向私有内部数据的引用——通过这些引用，调用者可以任意修改内部数据！请看下边的示例：

```
Point coord1(0, 0);
Point coord2(100, 100);

const Rectangle rec(coord1, coord2);  // rec 是一个 Rectangle 常量
                                      // 两顶点是(0, 0), (100, 100)

rec.upperLeft().setX(50);             // 但现在 rec 的两顶点却变为
                                      // (50, 0), (100, 100)!
```

upperLeft 返回了 rec 内部的 Point 数据成员，在这里请注意：虽然 rec 本身应该是 const 的，但是调用者竟可以使用 upperLeft 所返回的引用来修改这个数据成员！

上面的现象立刻引出了两个议题：首先，数据成员仅仅与访问限制最为宽泛的、返回该数据成员引用的函数拥有同等的封装性。在这种情况下，即使 ulhc 和 lrhc 声明为私有的，它们实际上仍然是公共的，这是因为公共函数 upperLeft 和 lowerRight 返回了指向它们的引用。其次，假设一个 const 成员函数返回一个引用，这一引用指向的数据与某个对象相关，但该数据却保存在该对象以外，那么函数的调用者就可以修改这一数据。（这仅仅是按位恒定限制所留下的隐患之一——参见条目 3。）

我们所做的一切都是围绕着返回引用的成员函数展开的，但是如果成员函数返回的是指针或者迭代器，同样的问题仍然会因为同样的理由发生。引用、指针、迭代器都可以称作“**句柄**”（获取其它对象的渠道），返回一个指向对象内部部件的句柄，通常都会危及到对象的封装性。就像我们看到的，即使成员函数是 const 的，返回对象的状态也是可以任意更改的。

大体上讲，对象的“内部部件”主要是它的数据成员，但是非公有成员函数（也就是 protected 和 private 的）同样也是对象的内部部件。与数据成员相同，返回指向成员函数的句柄也是糟糕的设计。这意味着你不应该让一个公有成员函数 A 返

回一个指向非公用成员函数 B 的指针。如果你这样做了，B 的访问权层次就与 A 一样了，这是因为客户将能够取得 B 的指针，然后通过这一指针来调用它。

所幸的是，返回指向成员函数指针的函数并不常见，所以我们还是把精力放在 Rectangle 类和它的 upperLeft 和 lowerRight 成员函数上。我们所发现的关于这些函数所存在的两个问题都可以简单的解决，只要将它们的返回值限定为 const 的就可以了：

```
class Rectangle {
public:
  ...
  const Point& upperLeft() const { return pData->ulhc; }
  const Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

使用这一改进的设计方案，客户就可以读取用来定义一个矩形的两个顶点，但是他们不可以修改这两个 Point。这就意味着将 upperLeft 和 lowerRight 声明为 const 并不是一个假象，因为它们将不允许调用者来修改对象的状态。至于封装问题，由于我们一直期望客户能够看到构造一个 Rectangle 的两个 Point，所以这里我们故意放松了封装的限制。更重要的是，这一放松是**有限的**：这些函数仅仅提供了读的访问权限。写权限仍然是禁止的。

即使这样，upperLeft 和 lowerRight 仍然会返回指向对象内部部件的句柄，而且会存在其他形式的问题。在某些特定的情况下，会导致悬空句柄：即引用对象中不再存在的某些部分的句柄。能够造成此类“会消失的对象”最普遍的东西就是函数返回值。举例说，请考虑以下函数，它以一个长方形的形式返回一个 GUI 对象的边界盒：

```
class GUIObject { ... };

const Rectangle boundingBox(const GUIObject& obj);
// 以传值方式返回一个矩形。关于返回值为什么是 const 的，请参见条目 3
```

现在请考虑一下客户可能怎样来使用这个函数：

```
GUIObject *pgo;                    // 让 pgo 指向某个 GUIObject
```

```
...

const Point *pUpperLeft = &(boundingBox(*pgo).upperLeft());
// 取得一个指向 boundingBox 左上角顶点的指针
```

调用 boundingBox 将会返回一个新的、临时的 Retangle 对象。这个对象没有名字，所以姑且叫它 temp。随后 temp 将调用 upperLeft，然后此次调用将返回一个指向 temp 内部部件的引用，特别地，指向构造 temp 的一个点。pUpperleft 将会指向这一 Point 对象。到目前为止一切都很完美，但是任务尚未完成，因为在这一语句的最后，boundingBox 的返回值（即 temp）将会被销毁，这样间接上会导致 temp 的 Point 被销毁掉。于是，pUpperLeft 将会指向一个并不存在的对象。这条语句创建了 pUpperLeft，可也让它成了悬空指针。

这就解释了为什么说：任何返回指向对象内部部件句柄的函数都是危险的。至于句柄是指针还是引用还是迭代器，函数是否是 const 的，成员函数返回的句柄本身是不是 const 的，这一切都无关紧要。只有一点，那就是：只要返回了一个句柄，那么就意味着你正在承担风险：它可能会比它指向的对象存活更长的时间。

这并不意味着你**永远也不能**让一个成员函数返回一个句柄。有些时候你不得不这样做。比如说，operator[]允许你获取 string 和 vector 中的任一元素，这些 operator[]的工作就是通过返回容器内部的数据来完成的（参见条目 3）——当容器本身被销毁时，这些数据同时也会被销毁。然而，这仅仅是一个例外，不是惯例。

时刻牢记

- 避免返回指向对象内部部件的句柄（引用、指针或迭代器）。这样做可以增强封装性，帮助 const 成员函数拥有更加恒定的行为，并且使悬空句柄出现的几率降至最低。

条目29： 力求代码做到“异常安全”

异常安全看上去像是孕育生命，但是让我们先把这种观点暂时放在一边。因为在一对恋人结婚之前，讨论生育问题还为时尚早。

假设我们正在设计一个表示 GUI 菜单的类，这种菜单是有背景图片的，由于这个类设计用于多线程环境中，因此它拥有一个互斥锁来确保正常的并发控制：

```
class PrettyMenu {
public:
  ...
  void changeBackground(std::istream& imgSrc);  // 更改背景图片
  ...

private:
  Mutex mutex;                                  // 本对象使用的互斥锁

  Image *bgImage;                               // 当前的背景图片
  int imageChanges;                             // 图片更改的次数
};
```

下面是 PrettyMenu 的 changeBackground 函数实现的一个备选方案：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  lock(&mutex);                     // 申请上锁(同条目 14)

  delete bgImage;                   // 删除旧的背景图片
  ++imageChanges;                   // 更新图片改变的次数
  bgImage = new Image(imgSrc);      // 装载新的背景图片

  unlock(&mutex);                   // 解锁
}
```

从异常安全的角度来说，这个函数简直一无是处。异常安全的两个基本要求，这个函数完全没有考虑到。

当抛出异常时，异常安全的代码应该做到：

- **不要泄漏资源。**上面的代码没有进行这项检测，这是因为如果“new Image(imgSrc)”语句产生了异常，那么对 unlock 的调用则永远不会兑现，这样该互斥锁将永远不会被解开。

- **不能让数据结构遭到破坏。**如果“new Image(imgSrc)”抛出异常，bgImage 就会指向一个已经销毁的对象。另外，无论新的图形是否装载成功，imageChanges 的数值都会增加。（从另一个角度说，旧的图形肯定是被删除了，那么你又怎么能确保图形被“改变”了呢。）

处理资源泄漏问题还是比较简单的，因为条目 13 中介绍过如何使用对象来管理资源，条目 14 中介绍过如何通过 Lock 类确保互斥锁在适当的时候被解开：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);                    // 来自条目 14 的经验：
                                      // 申请一个互斥锁，并确保适时解锁
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

能够让函数变得更短，是诸如 Lock 这样的资源管理类最让人兴奋的事情之一。你是否注意到：这里甚至不需要调用 unlock。更短的代码就是更优秀的代码，因为代码越短，它带来错误和误解的机会就越少。这是一条通用的准则。

把资源泄漏问题放在一旁，让我们把注意力集中在数据结构破坏的问题上。这里我们可以进行一次选择，但是在进行选择之前，我们首先要了解所需要的几个术语。

异常安全函数做出以下三种保证中的一项：

- 提供**基本保证**的函数会做出这样的承诺：如果抛出了一个异常，那么程序中的一切都将保持合法的状态。没有任何对象或数据结构会遭到破坏，所有的对象的内部都保持协调的状态（比如说，类所有的惯例都得到了满足）。然而，程序的具体状态可能是无法预知的。比如说，我们可以这样编写 changeBackground：如果某一时刻抛出一个异常，那么 PrettyMenu 对象可能继续使用旧的背景图片，也可以用某个默认的背景图片来代替。但是客户无法做出任何预测。（为了找到答案，客户大概会调用某个能明示当前背景图片的成员函数。）

- 提供**增强保证**的函数会做出这样的承诺：如果抛出了一个异常，那么函数的状态将保持不变。这样的函数仿佛是一个单一的原子，因为如果调用成功了，它就会大获全胜；一旦出了差错，程序状态将显示它从来没有被调用过。

  使用提供强保证的的函数要比使用仅提供基本保证的函数简单一些，这是因为在调用一个提供强保证的函数之后，程序只可能存在两种状态：一、按预期进行，函数成功执行；二、程序将保持函数调用前的状态。反观提供基本保证的函数，如果在调用它时抛出了一个异常，那么程序可能会处于**任何**合法的状态。

- 提供**零异常保证**的函数承诺程序决不会抛出异常，因为它们永远都会按部就班的运行。所有的内建数据类型（int、指针，等等）的操作都是零异常的（提供零异常保证）。这是异常安全代码必不可少的一个因素。

  我们可以假设包含空异常规范[①]的函数是不会抛出异常的，这样做看上去很合理，但是事实并不一定这样，请看下面的函数：

```
int doSomething() throw();          // 请注意：空异常规范
```

  这并不是说 doSomething 将永远不会抛出异常，这只是说，**如果** doSomething 抛出了异常，那么此时就出现了一个严重的错误，同时程序应该调用一个名为 unexpected 的函数[②]。实际上，doSomething 可能根本不会提供任何异常保证。函数的声明（包括它的异常规范，如果有的话）并不会告诉你它是否正确、是否小巧、是否高效，同时也不会告诉你它他提供了哪个层面上的异常安全保证。所有那些特性都要在函数实现中确定下来，而不是声明中。

异常安全的代码必须要提供上述三个层面的保证中的一种。否则它就不是异常安全的。那么你要做的就是：对于所写的每一个函数都要选择一个恰当层面的保证。除

---

① 异常规范是指：在函数声明时列出该函数可能抛出的异常的类型，并确保该函数不会抛出其它类型的异常。C++11 以后的版本已弃用这一特性，转而使用 noexpect 运算符来指定函数是否会抛出异常。——译者注

② 关于 unexpected 函数的更多信息，你可以参考你最喜欢的搜索引擎，或者“C++大全”一类的书籍。（搜寻 set_unexpected 可能会更好运些，因为它确定了 unexpected 的性质）

非我们正在处理异常不安全的老旧代码（这一点我们稍候再提）。只有当你的“优秀”的需求分析小组提出“你的程序需要泄露资源，并且需要使用破坏的数据结构”时，不提供异常安全保证也许才是一个可行的选择。

作为一个通用的准则，你会希望提供可行范围内最强的保证。从异常安全的角度来说，零异常的函数是美妙的，但是如果不去调用可能会抛出异常的函数，你是很难逾越 C++中 C 这一部分的。只要涉及动态内存分配（比如所有的 STL 容器），如果无法寻找到足够的内存来满足当前的要求，那么通常程序都会抛出一个 bad_alloc 异常（参见条目 49）。在可行的时候你应该为函数提供零异常保证，但是对于大多数函数而言，你需要在基本保证和增强保证之间做出选择。

对于 changeBackground，提供增强保证**似乎**并不是件难事。首先，我们可以改变 PrettyMenu 的 bgImage 数据成员的类型，从一个内建的 Image*指针类型转变为智能资源管理指针（参见条目 13）。坦白的说，单独从防止资源泄漏理论的角度上说，这是一个非常好的设计方案。事实上它简单地通过使用对象（比如智能指针）来管理资源（也就是遵循了条目 13 中的建议，这是优秀设计的基本要求），帮助我们提供了增强的异常安全保证。在下面的代码中，我将使用 tr1::shared_ptr，这是因为它的行为更直观，在进行复制操作时比 auto_ptr 更合适。

其次，我们从新编排了 changeBackground 中语句的顺序，从而使 imageChanges 直到图像改变以后才进行自加。作为一个通用的准则，直到一个事件真真切切地发生了，才去改变对象的状态来描述这个事件。

下面是改进后的代码：

```
class PrettyMenu {
  ...
  std::tr1::shared_ptr<Image> bgImage;
  ...
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);
```

```
  bgImage.reset(new Image(imgSrc)); // 将 bgImage 的内部指针替换为
                                    // "new Image"语句的结果
  ++imageChanges;
}
```

请注意这里不需要手动删除旧图片，因为这件事情完全由智能指针自行解决了。而且，只有在新图像成功创建之后，删除操作才会进行。更精确地说，tr1::shared_ptr::reset 函数只有在其参数（"new Image(imgSrc)"的结果）成功创建以后才会得到调用。由于只有在调用 reset 过程中才会使用 delete，因此如果从未进入该函数，就永远不会用到 delete。注意：使用对象（tr1::shared_ptr）来管理资源（动态分配的 Image），再次精简了 changeBackground。

如前所述，这两项改变**似乎**使 changeBackground 满足了增强的异常安全保证。可是白璧微瑕，imgSrc 参数还有一个小问题。如果 Image 的构造函数抛出了一个异常，那么输入流的读标记很可能已被移动，这样的移动可能会造成状态的变化，而这种变化对程序其它部分来说是可见的。在 changeBackground 解决这一问题之前，它仅仅提供基本的异常安全保证。

然而，让我们把这个问题暂时放在一旁，假装 changeBackground 确实可以提供增强保证。（我相信你可以想出一个办法来，可以通过改变参数的类型：从输入流变为包含图像信息的文件名。）有一个一般化的设计方案，可以使函数做到增强保证，了解这一方案十分重要。该方案一般称为"复制并 swap。"从理论上来讲，它非常简单。为需要修改的对象做一个副本，然后将所有需要做的改变应用于这个副本之上。如果期间任一个修改操作抛出了异常，那么原始的对象依然纹丝未动。在所有改变顺利完成之后，通过一次不抛出异常的操作将修改过的对象与原始对象相交换即可（参见条目 25）。

上述方案通常这样实现：将对象的所有数据从"真实的"对象复制到一个独立实现的对象中，然后为真实对象创建一个指针，将其指向这个实现对象。这通常称为"pimpl（pointer to implementation，指向实现的指针）惯用法"，条目 31 中将将解它的一些细节。对于 PrettyMenu 而言，典型的实现是这样的：

```
struct PMImpl {                          // PMImpl = PrettyMenu 的实现
```

```
  std::tr1::shared_ptr<Image> bgImage;     // 下文将介绍它为什么是结构体
  int imageChanges;
};

class PrettyMenu {
  ...

private:
  Mutex mutex;
  std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  using std::swap;                          // 参见条目 25

  Lock ml(&mutex);                          // 上锁

  std::tr1::shared_ptr<PMImpl>              // 复制对象数据
    pNew(new PMImpl(*pImpl));

  pNew->bgImage.reset(new Image(imgSrc));   // 修改副本
  ++pNew->imageChanges;

  swap(pImpl, pNew);                        // 交换新数据就位

}                                           // 解锁
```

在这个示例中，我做出了这样的选择：PMImpl 是一个结构体而不是类，由于 pImpl 是私有的，因此 PrettyMenu 数据的封装性得以保证。将 PMImpl 实现为类也不是不可以，但似乎便利性略差。（它同样会让面向对象的偏执狂们感到困惑。）如果需要，可以让 PMImpl 嵌套在 PrettyMenu 的内部，但是打包问题与编写异常安全代码的问题似乎没有什么联系，这不是我们当前所关注的。

有些修改对象状态的操作，要求要么是完全修改，要么完全不变，此时“复制并 swap”策略是完美的。但是，一般情况下，它并不能确保整个函数都做到增强保证。请看

下面 changeBackground 的一个抽象——someFunc，它使用了“复制并 swap”策略，但是它包含了两个其它函数（f1 和 f2）的调用：

```
void someFunc()
{
  ...                                  // 为本地的状态创建副本
  f1();
  f2();
  ...                                  // 交换修改后的状态就位
}
```

这里应该很清楚了：如果 f1 或者 f2 没有达到增强保证的要求，那么 someFunc 就很难满足增强保证。比如，假设 f1 仅提供了基本保证，为了让 someFunc 能满足增强保证，就必须要为其编写额外的代码，用于调用 f1 之前确定整个程序的状态，捕获 f1 抛出的所有异常，然后恢复原始的状态。

如果 f1 和 f2 都满足了增强保证，那么事情也不会好到哪去。如果 f1 运行完成，那么程序的状态可能经历了任意的修改过程，因此，一旦 f2 在此时抛出了一个异常，那么即使 f2 没有做任何修改操作，程序的状态也可能会与 someFunc 被调用时不一致。

问题在于函数的副作用。只要函数操作仅仅针对本地的状态（比如说，someFunc 仅仅影响到它所调用对象的状态），提供增强保证就相对简单些。当函数对于非本地数据存在副作用时，则更加困难些。比如说，如果调用 f1 引入的副作用是数据库被修改了，那么让 someFunc 满足异常安全的增强保证就比较困难。一般来说，已经被系统接受的数据库修改操作是无法撤销的，这是因为其它的数据库用户已经看到了数据库的新状态。

不管你情愿与否，诸如这样的问题会在你期望编写增强保证的函数时设置重重障碍。另一个问题是：效率。复制并 swap 策略的核心思想就是修改对象**副本**的数据，然后通过一个不会抛出异常的操作来交换修改后的数据。这需要为每个需要修改的对象创建出一个副本，这样做显然会浪费时间和空间，你也许不会情愿使用这一策略，现实条件有时也会阻止你。增强保证是我们良好的预期目标，只要可行你就应该提供，但是现实中它并不总是可行的。

在增强保证不可行时，你应该提供基本保证。从实用角度说，如果你发现你可以为某些函数提供增强保证，但是由此带来的效率和复杂度问题让增强保证变得得不偿失。只要你在必要的时候做出了努力使适当的函数满足了增强保证，那么对于一些函数仅提供基本保证就是无可厚非的。对于大多数函数而言，基本保证已经是合理的、完美的选择了。

如果你正在编写一个完全不提供异常安全保证的函数，那么就是另一番景象了。因为在这里完全可以在未证明你无罪之前假定你有罪。你本应该编写异常安全代码。但是你也可以为自己做出强有力的辩解。请再次考虑一下 `someFunc` 的实现，它调用了两个函数：`f1` 和 `f2`，假设 `f2` 完全没有提供异常安全保证，即使基本保证也没有，这就意味着一旦 `f2` 抛出一个异常，程序可能会在 `f2` 的内部发生资源泄露。这意味着 `f2` 中可能会有破损的数据结构，比如：排好序的数组可能不再按顺序排列，在两个数据结构之间转送的对象也可能会丢失数据，等等。这样 `someFunc` 也无力回天。如果 `someFunc` 函数调用了没有提供异常安全保证的函数，那么 `someFunc` 自身就无法做出任何保证。

让我们回到本节开篇时所说的“孕育生命”的问题。一位女性要么就是怀孕，要么就是没有，绝没有“部分怀孕”的状态。类似的，一个软件系统要么是异常安全的，要么就不是。没有所谓的“部分异常安全”的状态存在。在一个系统中，即使只有一个单独的函数不是异常安全的，那么整个系统也就不是异常安全的。遗憾的是，许多较为古老的 C++代码在编写的时候完全没有考虑到异常安全问题，因此当今许多系统便不是异常安全的。新系统中混杂着异常不安全的编写习惯。

没有理由去维持现状。当编写新代码或者修改现有代码的时候，要认真考虑一下如何使之做到异常安全。首先，使用对象管理资源。（依然参见条目 13。）这将有效地防止资源泄露。然后对于你要编写的每个函数确定你要使用哪一层面的异常安全保证，只有在调用古老的、没有异常安全保证的代码时才放弃异常安全保证，因为你别无选择。记录下你的选择，这即是为了你的客户，也是为了今后的维护人员。函数的异常安全保证位于接口的可见部分，因此你应该认真规划它，就像你认真规划接口其它部分一样。

四十年前，人们迷信充斥着 `goto` 的代码是完美的，现在我们却为了编写结构化控制流而努力。二十年前，全局的完全可访问的数据也是高踞神坛，然而当今我们却

在提倡封装数据。十年前，编写函数时不去考虑异常的影响的做法倍受追捧，但是今天，我们坚定不渝的编写异常安全代码。

岁月荏苒，我们在学习中不断进步……

时刻牢记

- 异常安全的函数即使在异常抛出时，也不会带来资源泄露，同时也不允许数据结构遭到破坏。这类函数提供基本的、增强的、零异常的三个层面的异常安全保证。
- 增强保证可以通过“复制并 swap”策略来实现，但是增强保证并不是对所有函数都适用。
- 函数所提供的异常安全保证通常不会强于其所调用函数中保证层次最弱的一个。

条目30： 深入探究内联函数

内联函数——多么**振奋人心**的一项发明！它们看上去与函数很相像，它们拥有与函数类似的行为，它们要比宏（参见条目 2）好用的多，同时你在调用它们时带来的开销比一般函数小得多。可谓“内联在手，别无他求。”

你得到的远远比你想象的要多，因为节约函数调用的开销仅仅是冰山一角。我们知道编译器优化通常是针对那些没有函数调用的代码，因此当你编写内联函数时，编译器就会针对函数体的上下文进行优化工作。大多数编译器都不会针对“外联”函数调用进行这样的优化。

然而，在你的编程生涯中，“没有免费的午餐”这句生活哲言同样奏效，内联函数也没有例外。内联函数背后蕴含的理念是：用代码本体来取代每次函数调用，这样做很可能会使目标代码的体积增大不少，这一点并不是非要统计学博士才能看得清。对于内存空间有限的机器而言，过分热衷于使用内联则会造成过多的空间占用。即使在虚拟内存中，那些冗余的内联代码也会带来不少无谓的分页，从而使缓存读取命中率降低，最终带来性能的牺牲。

另一方面，如果一个内联函数体**非常**的短，那么为函数体所生成代码的体积则可能会比调用函数所生成的代码小一些。此时，内联函数才真正做到了**减小**目标代码和提高缓存读取命中率的目的。

请时刻牢记，inline 是对编译器的一次**请求**，而不是一条命令。这种请求可以显式提出也可以隐式提出。隐式请求的途径就是：在类定义的内部定义函数：

```
class Person {
public:
  ...
  int age() const { return theAge; }   // 隐式内联请求:
  ...                                  // 年龄 age 在类定义中做出定义

private:
  int theAge;
};
```

这样的函数通常是成员函数，但是类中定义的函数也可以是友元（参见条目 46），如果函数是友元，那么它们会被隐式定义为内联函数。

显式声明内联函数的方法为：在函数定义之前添加 inline 关键字。比如说，下面是标准 max 模板（来自<algorithm>）常见的定义方式：

```
template<typename T>                   // 显式内联请求：
inline const T& std::max(const T& a, const T& b)
{ return a < b ? b : a; }              // 在 std::max 的前边添加"inline"
```

"max 是模板"这一事实，让我们不免得出这样的推论：内联函数和模板都应该定义在头文件中。这就使一些程序员做出"函数模板必须是内联函数"的论断。这一结论不仅站不住脚，而且也存在潜在的害处，所以这里还是要稍稍深入了解一下。

由于大多数编程环境的内联操作都是在编译过程中进行的，因此内联函数一般都应该定义在头文件中。编译器必须首先了解函数的情况，以便于用所调用函数体来代替这次函数调用。（一些构建环境在连接过程中进行内联，还有个别基于.NET 通用语言基础结构（CLI）的环境甚至是在运行时进行内联。这样的环境属于例外，不是守则。在大多数 C++程序中，内联是一个编译时行为。）

模板通常保存在头文件中，这是因为编译器需要了解这些模板，以便于在使用时进行正确的实例化。（再次说明，这并不是一成不变的。一些编程环境在连接时进行模板实例化。但是编译时实例化是更通用的方式。）

模板实例化相对于内联是独立的。如果你正在编写一个模板，而你又确信由这个模板所实例化出的所有函数都应该是内联的，那么这个模板就应该添加 `inline` 关键字；这也就是上文中 `std::max` 实现的做法。但是如果你正在编写的模板并不需要内联函数，那么就不要声明内联模板（无论是显式还是隐式）。内联也是有开销的，不计成本的引入内联并不明智。我们已经介绍过内联是如何使代码膨胀起来的（对于模板的作者而言，还应该做更周密的考虑——参见条目 44），但是内联还会带来其他的开销，这就是我们即将讨论的问题。

“`inline` 是对编译器的一次请求，但编译器可能会忽略它。”在我们的讨论开始之前，我们首先要弄清这一点。大多数编译器如果认为当前的函数过于复杂（比如包含循环或递归的函数），或者这个函数是虚函数（即使是最平常的虚函数调用），就会拒绝将其内联。后一个结论很好理解。因为 `virtual` 意味着“等到运行时再指出要调用哪个程序，”而 `inline` 意味着“在执行程序之前，使用要调用的函数来代替这次调用。”如果编译器不知道要调用哪个函数，那么它们拒绝内联函数体的做法就无可厚非了。

综上所述，我们得出下面的结论：一个给定的内联函数是否真正的得到内联，取决于你正在使用的构建环境——主要是编译器。幸运的是，大多数编译器拥有诊断机制，如果在内联某个函数时失败了，那么编译器将会做出警告（参见条目 53）。

有些时候，即使编译器认为某个函数非常适合内联，可是还是会为它生成一个函数体。举例说，如果你的程序要取得某个内联函数的地址，那么一般情况下编译器必须为其创建一个外联的函数体。编译器怎么能让一个指针去指向一个不存在的函数呢？再加上“编译器一般不会通过对函数指针的调用进行内联”这一事实，更能肯定这一结论：对于一个内联函数的调用是否应该得到内联，取决于这一调用是如何进行的：

```
inline void f() {...}              // 假设编译器乐意于将 f 的调用进行内联
void (*pf)() = f;                  // pf 指向 f
...
```

```
f();                                    // 此调用将被内联，因为这是一次“正常”的调用

pf();                                   // 此调用很可能不会被内联，
                                        // 因为它是通过一个函数指针进行的
```

即使你从未使用函数指针，未得到内联处理的内联函数依然会“阴魂不散”，这是因为调用函数指针的不仅仅是程序员。在对数组内的对象进行构造或析构的过程中，编译器有时会生成构造函数和析构函数的不恰当的版本，从而使它们可以得到这些函数的指针以便使用。

实际上，为构造函数和析构函数进行内联通常不是一个最佳选择。请看下面示例中 Derived 类的构造函数：

```
class Base {
public:
 ...
private:
   std::string bm1, bm2;            // 基类成员 1 和 2
};

class Derived: public Base {
public:
  Derived() {}                      // 派生类的构造函数为空 — 还有别的可能？
  ...
private:
  std::string dm1, dm2, dm3;        // 派生类成员 1-3
};
```

乍看上去，将这个构造函数进行内联再适合不过了，因为它不包含任何代码。但是你的眼睛欺骗了你。

C++对于在创建和销毁对象的过程中发生的事件进行了多方面的保证。比如，当你使用 new 时，你动态创建的对象就会通过构造函数将其自动初始化；当你使用 delete 时，将调用相关的析构函数。当你创建一个对象时，该对象的所有基类和所有数据成员将自动得到构造，在销毁这个对象时，相关的析构过程将会以反方向自动进行。如果在对象的构造过程中有异常抛出，那么对象中已经得到构造的部分

将统统被自动销毁。在所有这些场景中，C++告诉你**什么**一定会发生，但它没有说明**如何**发生。这一点取决于编译器的实现者，但是必须要清楚的一点是，这些事情并不是自发的。你必须要在程序中添加一些代码来实现它们。这些代码一定存在于某处，它们由编译器代劳，在编译过程中插入你的程序中。一些时候它们就存在于构造函数和析构函数中，所以，上文中 Derived 构造函数虽然看似是空的，但我们可以想象其与以下的具体实现中生成的代码是等价的：

```
Derived::Derived()                      // Derived"空"构造函数实现：概念版
{
 Base::Base();                          // 初始化 Base 部分

 try { dm1.std::string::string(); }     // 尝试构造 dm1
 catch (...) {                          // 如果抛出异常，
   Base::~Base();                       // 销毁基类部分，
   throw;                               // 并且传播该异常
 }

 try { dm2.std::string::string(); }     // 尝试构造 dm2
 catch(...) {                           // 如果抛出异常，
   dm1.std::string::~string();          // 销毁 dm1，
   Base::~Base();                       // 销毁基类部分，
   throw;                               // 并且传播该异常
 }

 try { dm3.std::string::string(); }     // 尝试构造 dm3
 catch(...) {                           // 如果抛出异常，
   dm2.std::string::~string();          // 销毁 dm2，
   dm1.std::string::~string();          // 销毁 dm1，
   Base::~Base();                       // 销毁基类部分，
   throw;                               // 并且传播该异常
 }
}
```

这段代码并不能完全反映出真实的编译器所做的事情，因为真实的编译器采用的做法更加精密。然而，上面的代码可以精确地反映出 Derived 的"空"构造函数必须

提供的内容。无论编译器处理异常的实现方式多么精密，Derived 的构造函数必须至少为其数据成员和基类调用构造函数，这些调用（可能就是内联的）会使 Derived 显得不那么适合进行内联。

这一推理过程对于 Base 的构造函数同样适用，因此如果将 Base 内联，所有添加进其中的代码同样也会添加进 Derived 的构造函数中（通过 Derived 构造函数调用 Base 构造函数的过程）。同时，如果 string 的构造函数恰巧是内联的，那么 Derived 的构造函数将为其复制出**五份副本**，分别对应 Derived 对象中包含的五个字符串（两个继承而来，另外三个系对象本身包括）。至于为什么“Derived 的构造函数是否该内联须深思熟虑”，现在可能就很容易理解了。对于 Derived 的析构函数也一样，无论如何，由 Derived 的构造函数所初始化的所有对象必须全部恰当的得到销毁。

库设计者必须估算出将函数内联所带来的影响，因为你无法为库中的客户可见的内联函数提供目标代码级别的升级。换句话说，如果 f 是库中的一个内联函数，那么库的客户就会将 f 的函数体编译进他们的程序中。如果一个库实现者在后期修改了 f 的内容，那么所有曾经使用过 f 的客户必须要重新编译。这一点是我们所不希望看到的。另一个角度讲，如果 f 不是内联函数，那么修改 f 只需要客户重新连接一下就可以了。这样要比重新编译减少很多繁杂的工作，并且，如果库中需要使用的函数是动态链接的，那么它对于客户就是完全透明的。

从开发优质程序的角度讲，这些重要问题应时刻牢记。但是以编写代码实际操作的角度来说，这一个事实将淹没一切：大多数调试人员面对内联函数时会遇到麻烦。这并不会令人意外，你如何为一个不存在的函数设定一个跟踪点呢？尽管一些构建环境提供了内联函数调试的支持，但是大多数环境都在调试过程中直接禁止内联。

对于“哪个函数应该声明为 inline 而哪些不应该”这一问题，我们可以由上文中引出一个逻辑上的策略。起初，不要内联任何内容，或者仅挑选出那些不得不内联的函数（参见条目 46）或者那些确实是很细小的程序（比如本节开篇处出现的 Person::age）进行内联。谨慎引入内联，你就为调试工作提供了方便，但是你仍然要为内联摆正位置：它属于手工的优化操作。不要忘记 80-20 经验决定主义原则：一个典型的程序将花去 80%的时间仅仅运行 20%的代码。这是一个非常重要的原则，因为它提醒着我们软件开发者的目标：找出你的代码中 20%的这部分进行优

化，从而从整体上提高程序的性能。你可以花费漫长的时间进行内联、修改函数等等，但如果你没有锁定正确的目标，那么你做再多的努力也是徒劳。

时刻牢记

- 仅仅对小型的、调用频率高的程序进行内联。这将简化你的调试操作，提供更灵活的目标代码可更新性，降低潜在的代码膨胀发生的可能，并且可以让程序获得更高的速度。
- 不要将模板声明为 `inline` 的，因为它们一般在头文件中出现。

条目31： 尽量减少文件间的编译依赖

你在开发工作中，可能需要对于某个类的具体实现做出一个细小的修改。提醒你一下，修改的地方不是类接口，而是实现本身，并且仅仅是私有部分。完成之后，你开始对程序进行重新构建，这时你肯定会认为这一过程将十分短暂，毕竟你只对一个类做出了修改。当你按下“构建”按钮，或输入 `make` 命令（或者其他什么等价的操作）之后，你惊呆了，然后你脸上便呈现出一个大大的囧字，因为你发现**整个世界**都重新编译并重新链接了！真是人神共**愤**！

问题的症结在于：C++并不擅长区分接口和实现。一个类的定义不仅指定了类接口的内容，而且指明了相当数量的实现细节。请看下面的示例：

```
class Person {
public:
  Person(const std::string& name, const Date& birthday,
         const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...

private:
     std::string theName;          // 具体实现
     Date theBirthDate;            // 具体实现
```

```
    Address theAddress;          // 具体实现
};
```

这里，如果无法访问 Person 具体实现所使用的类（即 string、Date 和 Address）定义，那么 Person 类将不能够得到编译。通常这些定义通过#include 指令来提供，因此在定义 Person 类的文件中，你应该能够找到这样的内容：

```
#include <string>
#include "date.h"
#include "address.h"
```

不幸的是，这样做使得定义 Person 的文件对这些头文件产生了依赖。如果任一个头文件的内容被修改了，或者这些头文件所依赖的**另外**某个头文件被修改，那么包含 Person 类的文件就必须重新编译，有多少个文件包含 Person，就要进行多少次编译操作。这种层叠式的编译依赖将招致无法估量的灾难式后果。

你可能会考虑：为什么 C++坚持要将类具体实现的细节放在类定义中呢？假如说，如果我们换一种方式定义 Person，单独编写类的具体实现，结果又会怎样呢？

```
namespace std {
    class string;                // 前置声明（这个是非法的，参见下文）
}

class Date;                      // 前置声明
class Address;                   // 前置声明

class Person {
public:
    Person(const std::string& name, const Date& birthday,
           const Address& addr);
    std::string name() const;
    std::string birthDate() const;
    std::string address() const;
   ...
};
```

如果这样可行，那么对于 Person 的客户来说，仅在类接口有改动时，才需要进行重新编译。

这种想法存在着两个问题。首先，string 不是一个类，它是一个 typedef（typedef basic_string<char> string）。于是，针对 string 的前置声明就是非法的。实际上恰当的前置声明要复杂的多，因为它涉及到其他的模板。然而这不是主要问题，因为你本来就不应该尝试手工声明标准库的内容。仅仅使用恰当的#include 指令就可以了。标准头文件一般都不会成为编译中的瓶颈，尤其是在你的编译环境允许你利用预编译的头文件时更为突出。如果分析标准头文件对你来说的确是件麻烦事，那么你可能就需要改变你的接口设计，以避免使用那些会带来多余#include 指令的标准类成员。

对所有的类做前置声明会遇到的第二个（同时也是更显著的）难题是：在编译过程中，编译器需要知道对象的大小。请观察下面的代码：

```
int main()
{
 int x;                              // 定义一个 int

 Person p( 参数表 );                 // 定义一个 Person
   ...
}
```

当编译器看到了 x 的定义时，它们就知道该为其分配足够的内存空间（通常位于栈中）以保存一个 int 值。这里没有问题。每一种编译器都知道 int 的大小。当编译器看到 p 的定义时，他们知道该为其分配足够的空间以容纳一个 Person，但是他们又如何得知 Person 对象的大小呢？得到这一信息的唯一途径就是通过类定义，但是如果省略类定义具体实现细节是合法的，那么编译器又如何得知需要分配多大空间呢？

同样的问题不会在 Smalltalk 和 Java 中出现，因为在这些语言中，每当定义一个对象时，编译器仅仅分配指向该对象**指针**大小的空间。也就是说，在这些语言中，上面的代码将做如下的处理：

```
int main()
{
```

```
  int x;                                // 定义一个 int

  Person *p;                            // 定义一个 Person
  ...
}
```

当然，这段代码在 C++中是合法的，于是你可以自己通过“将对象实现隐藏在指针之后”来玩转前置声明。对于 Person 而言，实现方法之一就是将其分别放在两个类中，一个只提供接口，另一个存放接口对应的具体实现。暂且将具体实现类命名为 PersonImpl，Person 类的定义应该是这样的：

```
#include <string>                       // 标准库成员，不允许对其进行前置声明
#include <memory>                       // 为使用 tr1::shared_ptr; 稍后介绍

class PersonImpl;                       // Person 实现类的前置声明

class Date;                             // Person 接口中使用的类的前置声明
class Address;

class Person {
public:
 Person(const std::string& name, const Date& birthday,
        const Address& addr);
 std::string name() const;
 std::string birthDate() const;
 std::string address() const;
 ...

private:                                // 指向实现的指针
  std::tr1::shared_ptr<PersonImpl> pImpl;
};                                      // 关于 std::tr1::shared_ptr 的更多信息
                                        // 参见条目 13
```

在这里，主要的类（Person）仅仅包括一个数据成员——一个指向其实现类（PersonImpl）的指针（这里是一个 tr1::shared_ptr，参见条目 13）。我

们通常将这样的设计称为 **pimpl 惯用法**（指向实现的指针）。在这样的类中，指针名通常为 pImpl，就像上面代码中一样。

通过这样的设计，Person 的客户将会与日期、地址和人这些具体信息隔离开。你可以随时修改这些类的具体实现，但是 Person 的客户不需要重新编译。另外，由于客户无法得知 Person 的具体实现细节，他们就不容易编写出依赖于这些细节的代码。这样做真正起到了分离接口和实现的目的。

这项分离工作的关键所在，就是用声明的依赖来取代定义的依赖。这就是最小化编译依赖的核心所在：只要可行，就要将头文件设计成自给自足的，如果不可行，那么就依赖于其他文件中的声明语句，而不是定义。其他一切事情都应遵从这一基本策略。于是有：

- 只要使用对象的引用或指针可行时，就不要使用对象。只要简单地通过类型声明，你就可以定义出类型的引用和指针。反观定义类型对象的情形，你就必须要进行类型定义了。
- 只要可行，就用类声明依赖的方式取代类定义依赖。请注意，如果你需要声明一个函数，该函数会使用某个类，那么在任何情况下类的定义都不是必须的。即使这个函数以传值方式传递或返回这个类的对象：

```
class Date;                          // 类声明

Date today();                        // 这样是可行的
void clearAppointments(Date d);      // 没有必要对 Date 类做出定义
```

当然，传值方式在通常情况下都不会是优秀的方案（参见条目 20），但是如果你出于某种原因使用了传值方式时，此时必将引入不必要的编译依赖，你依然难择其咎。

即使不定义 Date 的具体实现，today 和 clearAppointments 依然可以正确声明，C++的这一能力可能会让你感到吃惊，但是实际上这一行为又没有想象中那么古怪。如果代码中任意一处调用了这些函数，那么在这次调用前的某处必须要对 Date 进行定义。此时你又有了新的疑问：为什么我们要声明没有人调用的函数呢，这不是多此一举吗？这一疑问的答案很简单：这种函数并不是没有人调用，而是不是所有人都会去调用。假设你的库中包含许多函数声明，这并不意味着每一位客户

都会使用到所有的函数。上文的做法中，提供类定义的职责将从头文件中的函数声明转向客户端文件中包含的函数调用，通过这一过程，你就排除了手工造成的客户端类定义依赖，这些依赖实际上是多余的。

- 为声明和定义分别提供头文件。为了进一步贯彻上文中的思路，头文件必须要一分为二：一个存放声明，另一个存放定义。当然这些文件必须保持相互协调。如果某处的一个声明被修改了，那么相应的定义处就必须做出相应的修改。于是，库的客户就应该始终使用#include 指令来包含一个声明头文件，而不是自己进行前置声明，库的作者应提供两个头文件。比如说，在 Date 的客户期望声明 today 和 clearAppointments 时，就应该无需向上文中那样，对 Date 进行前置声明。更好的方案是用#include 指令来引入恰当的声明头文件：

```
#include "datefwd.h"              // 包含 Date 类声明(而不是定义)的头文件

Date today();                     // 同上
void clearAppointments(Date d);
```

头文件“datefwd.h”中仅包含声明，这一名字基于 C++标准库中的<iosfwd>（参见条目 54）。<iosfwd>包含着 IO 流组件的声明，这些组件相应的定义分别存放在不同的几个头文件中，包括：<sstream>、<streambuf>、<fstream>以及<iostream>。

从另一个角度来讲，使用<iosfwd>作示例还有一定的示范效应，因为它告诉我们本节中的建议不仅对非模板的类有效，而且对模板同样适用。尽管在条目 30 中分析过，在许多构建环境中，模板定义通常保存在头文件中，一些构建环境中还是允许将模板定义放置在非头文件的代码文件里，因此提供为模板提供仅包含声明的头文件并不是没有意义的。<iosfwd>就是这样一个头文件。

C++提供了 export 关键字，它用于分离模板声明和模板定义。但是遗憾的是，支持 export 的编译器是十分有限的，现实中 export 的应用更是寥寥无几。因此在高效 C++编程中，export 究竟扮演什么角色，讨论这个问题还为时尚早。

诸如 Person 此类使用 pimpl 惯用法的类通常称为**句柄类**。为了避免你对这样的类如何工作产生疑问，一个途径就是将类中所有的函数调用放在相关的具体实现类

之前，并且让这些具体实现类去做真实的工作。请看下面的示例，其中演示了 Person 的成员函数应该如何实现：

```
#include "Person.h"                    // 我们将编写 Person 类的具体实现，
                                       // 因此此处必须包含类定义。

#include "PersonImpl.h"                // 同时，此处必须包含 PersonImpl 的类定义，
                                       // 否则我们将不能调用它的成员函数；请注意，
                                       // PersonImpl 拥有与 Person 完全一致的成员
                                       // 函数 - 也就是说，它们的接口是一致的。

Person::Person(const std::string& name, const Date& birthday,
               const Address& addr)
: pImpl(new PersonImpl(name, birthday, addr))
{}

std::string Person::name() const
{
  return pImpl->name();
}
```

请注意下面两个问题：Person 的构造函数是如何调用 PersonImpl 的构造函数的（通过使用 new——参见条目 16），以及 Person::name 是如何调用 PersonImpl::name 的。这两点很重要。将 Person 定制为一个句柄类并不会改变它所做的事情，而是仅仅改变它做事情的方式。

除了句柄类的方法，我们还可以采用一种称为“**接口类**”的方法来将 Person 定制为特种的抽象基类。这种类的目的就是为派生类指定一个接口（参见条目 34）。于是，通常情况下它没有数据成员，没有构造函数，但是拥有一个虚析构函数（参见条目 7），以及一组指定接口用的纯虚函数。

接口类与 Java 和.NET 中的接口一脉相承，但是 C++并没有像 Java 和.NET 中那样对接口做出非常严格的限定。比如说，无论是 Java 还是.NET 都不允许接口中出现数据成员或者函数实现，但是 C++对这些都没有做出限定。C++拥有更强灵活性是有实用价值的。就像条目 36 中所解释的那样，由于非虚函数的具体实现对于同一

层次中所有的类都应该保持一致，因此不妨将这些函数实现放置在声明它们的接口类中，这样做是有意义的。

Person 的接口类可以是这样的：

```
class Person {
public:
  virtual ~Person();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
  virtual std::string address() const = 0;
  ...
};
```

这个类的客户必须要基于 Person 的指针和引用来编写程序，因为实例化一个包含纯虚函数的类是不可能的。（然而，实例化一个**继承**自 Person 的类却是可行的——参见下文。）就像句柄类的客户一样，接口类客户除非遇到接口类的接口有改动的情况，其他任何情况都不需要对代码进行重新编译。

接口类的客户必须有一个创建新对象的手段。通常情况下，它们可以通过调用真正被实例化的派生类中的一个函数来实现，这个函数扮演的角色就是派生类的构造函数。这样的函数通常被称作工厂函数（参见条目 13）或者**虚构造函数**。这种函数返回一个指向动态分配对象的指针（最好是智能指针——参见条目 18），这些动态分配的对象支持接口类的接口。这样的函数通常位于接口类中，并且声明为 static 的：

```
class Person {
public:
 ...

 static std::tr1::shared_ptr<Person>   // 返回一个 tr1::shared_ptr，
   create(const std::string& name,     // 它指向一个 Person 对象，这个
          const Date& birthday,        // Person 对象由给定的参数初始化，
          const Address& addr);        // 为什么返回智能指针参见条目 18
 ...
```

```
};
```

客户这样使用：

```
std::string name;
Date dateOfBirth;
Address address;
...

// 创建一个支持 Person 接口的对象
std::tr1::shared_ptr<Person>
                    pp(Person::create(name, dateOfBirth, address));

...

std::cout << pp->name()             // 通过 Person 的接口使用这一对象
          << " 出生日期 "
          << pp->birthDate()
          << " 家庭住址 "
          << pp->address();
...                                 // 当程序执行到 pp 的作用域之外时，
                                    // 这一对象将被自动删除——参见条目 13
```

当然，与此同时，必须要对支持某一接口类的接口的具体类做出定义，并且必须有真实的构造函数得到调用。比如说，有一个具体的派生类 `RealPerson` 使用了接口类 `Person`，这一派生类应为其继承而来的虚函数提供具体实现：

```
class RealPerson: public Person {
public:
  RealPerson(const std::string& name, const Date& birthday,
             const Address& addr)
  : theName(name), theBirthDate(birthday), theAddress(addr)
  {}

  virtual ~RealPerson() {}

  std::string name() const;         // 这里省略了这些函数的具体实现，
```

```
  std::string birthDate() const;   // 但是很容易想象它们是什么样子。
  std::string address() const;

private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
};
```

有了 RealPerson，编写 Person::create 便手到擒来：

```
std::tr1::shared_ptr<Person> Person::create(const std::string& name,
                                            const Date& birthday,
                                            const Address& addr)
{
  return std::tr1::shared_ptr<Person>(
         new RealPerson(name, birthday,addr));
}
```

Person::create 还可以以一个更加贴近现实的方法来实现，它应能够创建不同种类的派生类对象，创建的过程基于某些相关信息，例如 :新加入的函数的参数值、从一个文件或数据库中得到读到的数值，环境变量，等等。

RealPerson 向我们展示了实现接口类的两种最通用的实现机制之一：它的接口规范继承自接口类（Person），然后实现接口中的函数。第二种实现接口类的方法牵扯到多重继承，那是条目 40 中探索的主题。

句柄类和接口类将接口从实现中分离开来，因此降低了文件间的编译依赖。如果你喜欢吹毛求疵，那么你一定在等待我来添加一条注释。“这么多变魔术般古怪的事情会带来多大开销？”这个问题的答案就是计算机科学中极为普遍的一个议题：你的程序在运行时更慢了一步，另外，每个对象所占的空间更大了一点。

使用句柄类的情况下，成员函数必须通过实现指针来取得对象的数据。这样无形中增加了每次访问时迂回的层数。同时，为保证每个对象都拥有足够的内存空间，你必须增大实现指针所指向的区域的内存空间。最后，你必须要对实现指针进行初始化（在句柄类的构造函数中），以便于将其指向一个动态分配的实现对象，这样做

将会招致动态内存分配（以及由此产生的释放）所带来的固有的开销，也有可能遭遇 `bad_alloc`（内存不足）异常。

由于对于接口类来说每次函数调用都是虚拟的，因此你在每调用一次函数的过程中你就会为其付出一次间接跳转的代价（参见条目 7）。同时，派生自接口类的对象必须包含一个虚函数表指针（依然参见条目 7）。这一指针也可能会加大保存一个对象所需要的空间，这取决于接口类是否是该对象中虚函数的唯一来源。

最后，无论是句柄类还是接口类，都不适合于过多使用内联。条目 30 中解释了为什么一般情况下要将内联的函数体放置在头文件中，然而句柄和接口类都是特别设计用来隐藏诸如函数体等具体实现内容的。

然而，对于句柄类和接口类来说，仅仅由于它们会带来一些额外的开销就远离它们，也是一个严重的错误。你并不会因为虚函数存在缺点放弃使用它，不是吗？（如果你真的想放弃，那么你可能看错书了。）你应该以一个进化演进的方式来使用这些技术。在开发过程中，尝试使用句柄类和接口类来减少改动具体实现时为客户带来的影响。在生产环境中，如果应用这些技术导致程序的速度和/或大小的变动足够显著，从而冲淡了不同的类之间所增加关系度的影响，应适时使用具体的类来取代句柄类和接口类。

时刻牢记

- 最小化编译依赖的基本理念就是使用声明依赖代替定义依赖。基于这一理念有两种实现方式，它们是：句柄类和接口类。
- 库的头文件必须以完整、并且仅存在声明的形式出现。无论是否涉及模板。

第六章. 继承和面向对象设计

面向对象的程序设计（OOP）风靡计算机软件界已经有近 20 个年头了，因此，你或多或少会与继承、派生以及虚函数这些事物有所接触。即使你仅仅使用 C 语言编程，那你也难以彻底逃离 OOP 的大气候。

然而，C++中的 OOP 与你过去常见的可能有所不同。继承可以是单一的也可以是多重的，同时每个继承链接可以是公有的(public)，受保护的(protected)，或者私有的(private)。每个连接也可以分为虚拟(virtual)和非虚拟两种。因此成员函数也有相关的选项：虚拟的？非虚拟的？纯虚拟的？此外，与其他语言特征进行交互也有需要考虑的问题：默认参数值是如何与虚函数相交互的？继承是如何影响 C++的名字查找规则的？设计有哪些选项？如果需要将一个类的行为设计成可修改的，那么虚函数是实现这一特性的最优途径吗？

本章就会针对这些问题为大家一一道来。而后，我还将向大家解释 C++种特殊功能的真正**含义**。——也就是当你使用某一特定的思路时，你真正**表达**出的内容。比如说，公共继承意味着“A 是一个 B”，倘若你让其表示任何其它的含义，那么你就会惹上麻烦。类似地，一个虚函数意味着“接口必须被继承”，然而一个非虚函数则意味着“接口和实现必须都被继承。”一个 C++程序员如果不能恰当的区分这些内容的含义，那么他将步履维艰。

如果你能够深入了解 C++浩瀚的特性，你将发现你对 OOP 的见解将有所改变。你的见解将决定你对软件系统的整体认识。我们不去在对语言特性的区分上做过多纠结，而是将中心放在“对于你的软件系统，你想说些什么”上。一旦你心里真正的清楚你要说的话，那么将你的思路翻译为 C++代码也没有那么费力。

条目32： 确保公共继承以“A 是一个 B”形式进行

威廉 德门特在他的著作《几人留守几人眠》(W. H. Freeman and Company, 1974)中讲述了这样一个故事：他在课堂上尝试让他的学生记住课程中最重要的那一部分。他和他的学生讲，据说普通的英国学生仅仅能记得黑斯廷斯战役发生于 1066 年。德门特强调，如果有一个学生只记住了一点点，他（她）记住的便是 1066 这个年号。德门特继续讲，对于**他的**课堂上的学生，只有几条核心的信息，其中还包括一

条有趣的结论——“安眠药最终会导致失眠”。他请求他的学生们一定要记住这些核心信息，即使把课堂中讨论的所有其他的内容都忘光也可以，整个学期他都为学生反复重复这些核心信息。

在学期末，期末考试的最后一道题是“请写下这一学期中让你铭记一生的一件东西。”当德门特阅卷的时候，差点儿没昏过去。几乎所有的学生不约而同地写下了“1066”。

因此，我“诚惶诚恐”地向各位讲述 C++面向对象编程中最为重要（没有之一）的一条原则：公共继承意味着“A 是一个 B”关系。这条原则一定要深深的印在脑子里。

如果你编写了 B 类（Base，基类），并编写了由其派生出的 D 类（Derived，派生类），那么你就告诉了 C++编译器（以及代码的读者），每一个 D 类型的对象同时也是 B 类型的，但是**反过来不成立**。你要表达的是：B 表示比 D 更加一般化的内容，而 D 则表示比 B 更加具体化的内容。另外我们还强调如果一个 B 类型的对象可以在某处使用时，那么 D 类型的对象一定可以在此使用。这是因为 D 类型的每个对象**一定是** B 类型的。反之，如果某一刻你需要一个 D 类型的对象，那么一个 B 类型的对象则不一定能满足要求：每个 D 都是一个 B，但反之不然。

C++严格按上述方式解释公共继承。请参见下面的示例：

```
class Person {...};
class Student: public Person {...};
```

我们从生活的经验中可以得知：每个学生都是一个人，但是并不是每个人都是学生。上面的代码精确的体现了这一层次结构。我们期望“人”的每一条属性对“学生”都适用（比如人有出生日期，学生也有）。但是对学生能成立的属性对于一般的人来说并不一定成立（比如一个学生被某所大学录取了，但不是每个人都会去上大学）。人的概念比学生的概念更加宽泛，而学生是一类特殊的人。

在 C++领域中，一切需要使用 Person 类型参数（或指向 Person 的指针或引用）的函数同样能够接受 Student 对象（或指向 Student 的指针或引用）：

```
void eat(const Person& p);          // 人人都会吃饭

void study(const Student& s);       // 只有学生会学习
```

```
Person p;                          // p 是一个人
Student s;                         // s 是一个学生

eat(p);                            // 正确，p 是一个人

eat(s);                            // 正确，s 是一个学生，
                                   // 同时一个学生是一个人

study(s);                          // 正确

study(p);                          // 错误！p 不一定是学生
```

这一点仅仅在**公共继承**的情况下成立。只有在 Student 是由 Person 公共派生而来时，C++才会按刚才描述的情景运行。私有继承将引入一个全新的议题（我们在条目 39 中讨论）。受保护的继承我至今也没有弄明白，暂且不谈。

公共继承和“A 是一个 B”的等价性听上去很简单，但是有时你的直觉会误导你。比如说，“企鹅是鸟”这是千真万确的，同时，“鸟会飞”也是不争的事实。如果我们在C++中如此幼稚地表述这一情景，那么我们将得到：

```
class Bird {
public:
  virtual void fly();              // 鸟会飞
  ...
};

class Penguin:public Bird {        // 企鹅是鸟
  ...
};
```

瞬间我们陷入泥潭，因为这一层次结构中，企鹅竟然会飞！这显然是荒谬的。那么问题出在哪里呢？

这种情况下，我们成为了一种不精确的语言——英语的受害者。当我们说“鸟类能够飞行”时，我们的意思并不是说**所有的**鸟类都会飞。在一般情况下，只有拥有飞行能力的鸟类才能够飞。假如我们的语言更加精确些，我们就能认识到世界上还存

在着一些不会飞的鸟类，我们也就能构建出下面的层次结构，这样的机构才更加贴近真实世界：

```
class Bird {
  ...                                  // 不声明任何飞行函数
};

class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};

class Penguin: public Bird {

  ...                                  // 不声明任何飞行函数

};
```

这一层次结构比原先设计的更加忠实于我们所了解的世界。

到目前为止，上文的飞禽问题尚未彻底明了，因为在一些软件系统中，区分鸟类是否可以飞行这项工作是没有意义的，如果你的程序主要是关于鸟类的喙和翅膀，而与飞行没有什么关系，那么原先的 2 个类的层次结构就可以满足要求了。这里也很清晰的反映出了这一哲理：凡事并不存在一劳永逸的解决方案。对软件系统而言，最好的设计一定会考虑到这个系统是用来做什么的，无论是现在还是未来，如果你的程序对飞行的问题一无所知，并且也不准备去了解，那么忽略飞行特性的设计方案很可能就是完美的。事实上，这样做要比将两者区分开的设计方案更好些，因为你正在模拟的世界中很可能不会存在这一机制。

对于解决上文中的“白马非马”问题，还存在另外一个思考方法。那就是为企鹅重新定义 `fly` 函数，从而让其产生一个运行时错误：

```
void error(const std::string& msg);     // 定义的内容在其他地方

class Penguin: public Bird {
```

```
public:
  virtual void fly() { error("尝试让一只企鹅飞行！");}

  ...

};
```

一定要认识到：这样做不一定能达到预期效果。因为这并不是说“企鹅不会飞”，而是说“企鹅会飞，但是在它尝试飞行时出错了”。这一点很重要。

如何找出两者的区别呢？我们从捕获错误的时机入手：“企鹅不会飞”的指令可以由编译器做出保证，但是对于“企鹅在尝试飞行时出错”这一规则的违背只能够在运行时捕获。

为了表达这一约束，“企鹅不能飞行——**句号**”，你要确认企鹅对象一定没有飞行函数定义：

```
class Bird {
  ...                                  // 不声明任何飞行函数
};

class Penguin: public Bird {
  ...                                  // 不声明任何飞行函数
};
```

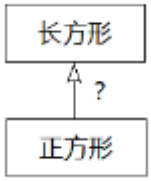
现在，如果你尝试让一个企鹅飞行，那么编译器将对你的侵犯行为做出抗议：

```
Penguin p;
p.fly();                               // 错误！
```

如果你适应了“产生运行时错误”的方法，上文代码的行为则与你所了解的大相径庭。使用上文中的方法，编译器不会对 `p.fly` 的调用做出任何反应。条目 18 中解释了好的接口设计能够防止非法代码得到编译。因此你最好使用在编译室拒绝企鹅尝试飞行的设计方案，而不是仅仅在运行时捕获错误。

可能你承认你的鸟类学知识并不丰富，但对于初级几何你还是有信心的吧，让我们拿长方形和正方形再举一个例子，这没有什么复杂的吧？

好，请回答一个简单的问题：Square（正方形）类是否应该公共继承自 Rectangle（长方形）类？



你会说："当然可以了！正方形就是一个长方形，这地球人都知道。但反过来就不成立了。"这一点至少在学校里是正确的，但我们都已经不是小学生了，请考虑下面的代码：

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height() const;      // 返回当前值
  virtual int width() const;

  ...
};

void makeBigger(Rectangle& r)      // 增加 r 面积的函数
{
  int oldHeight = r.height();

  r.setWidth(r.width() + 10);      // 为 r 的宽增加 10

  assert(r.height() == oldHeight); // 断言 r 的高不变
}
```

显然地，这里的判断永远不会失败，makeBigger 仅仅改变了 r 的宽，它的高始终没有改变。

现在请观察下面的代码，其中使用了公共继承，从而使得正方形得到与长方形一致的待遇：

```
class Square: public Rectangle {...};

Square s;
...

assert(s.width() == s.height());   // 这对所有的正方形都成立

makeBigger(s);                     // 根据继承关系，s 是一个长方形
                                   // 因此我们可以增加它的面积

assert(s.width() == s.height());   // 对所有的正方形也应成立
```

第二次判断同样不应该出错，这也是十分明显的。因为正方形的定义要求长宽值永远相等。

但是现在我们又遇到了一个问题，我们如何协调下面的断言呢？

- 在调用 makeBigger 之前，s 的高与宽相等；
- 在 makeBigger 内部，s 的宽值该变了，但是高没有；
- 从 makerBigger 返回后，s 的高与宽又相等了。（请注意：s 是通过引用传入 makeBigger 中的，因此 makeBigger 改变的是 s 本身，而不是 s 的副本。）

欢迎来到公共继承的美妙世界。在这里，你在其他领域（包括数学）所积累的经验也许不会按部就班地奏效。这种情况下最基本的问题就是：一些对长方形可用的属性（它的长、宽可以分别修改），对于正方形而言并不适用（它的长、宽必须保持一致）。但是，公共继承要求对基类成立的一切对于派生类同样应该能够成立——一切的**一切**！这种情况下，长方形和正方形（以及条目 38 中集合、线性表）的实例都会遇到问题。因此，使用公共继承来构建它们之间的关系显然是错误的。编译器会允许你这样做，但是就像我们刚刚所看到的一样，我们无法确保代码是否能够

按要求运行。这件事每一位程序员一定深有体会(往往比其他行业的人要深得多),因为许多情况下代码能够通过编译并不意味着它能够正常运行。

在我们投入面向对象程序设计的怀抱时,多年积累的编程经验难道成为了我们的绊脚石吗?这一点你无需顾虑。旧有的知识依然是宝贵的,只是既然你已经把继承的概念添加进你大脑中的设计方案库中,你就应该以全新的眼光来开拓自己的感官世界,从而使你在面对包含继承的程序时不会迷失方向。假如有人向你展示了一个几页长的程序,你也可以从企鹅继承自鸟类、正方形继承自长方形这些示例所包含的理念中,找出同样有趣的东西。这**有可能**是完成工作的正确途径,只是这个可能性并不大。

类间的关系并不仅限于"A 是一个 B"关系。另外还存在两个内部类关系,它们是:"A 拥有一个 B"、"A 是以 B 的形式实现的"。这些关系将在条目 38 和条目 39 中讲解。由于人们往往会将上面两种关系其中之一错误地构造成"A 是一个 B",因此随之带来的 C++设计错误比比皆是,所以你应该确保对于这些关系之间的区别有着充分的理解,这样你才能在 C++中分别对这些关系做出最优秀的构造。

时刻牢记

- 公共继承意味着"A 是一个 B"的关系。对于基类成立的一切都应该适用于派生类,因为派生类的对象就是一个基类对象。

条目33: 避免隐藏继承而来的名字

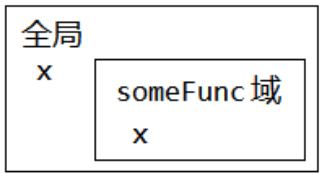
莎士比亚对于"名字"有着独特的见解。"名字意味着什么?玫瑰不叫玫瑰,依然芬芳如故。"大师还写道:"倘若有人偷窃了我的好名字……事实上会让我变得一贫如洗。"让这两段名句引领我们去探究 C++中继承的名字。

事实上,本节讨论的问题与继承并没有太大关系。它仅仅关系到作用域。我们都能读懂下面的代码:

```
int x;                              // 全局变量

void someFunc()
{
```

```
  double x;                              // 局部变量

  std::cin >> x;                         // 读一个新值赋给局部变量 x
}
```

为 x 赋值的语句是关于局部变量 x 的，而不是全局变量 x，这是因为内部作用域隐藏了（“遮挡了”）外部作用域的名字。我们可以将这种域间状况用下图描述：

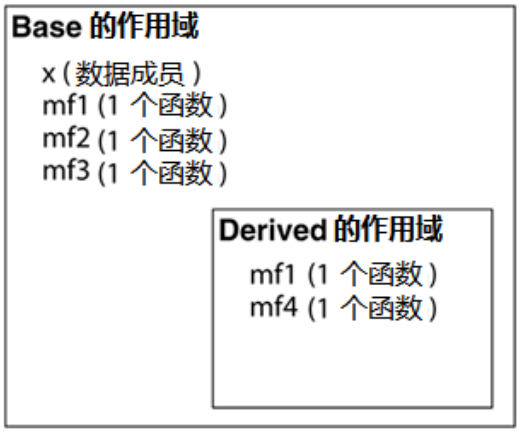


当编译器执行至 someFunc 的作用域内并且遇到名字 x 时，它将在局部作用域内查找，以便确认此处是否包含与 x 这个名字相关的操作。因为如果有的话，编译器就不会再去检查其它任何作用域了。在这上面的示例中，someFunc 中的 x 是 double 类型的，全局变量 x 是 int 类型的，但是这无关紧要，C++的名字隐藏准则只会做一件事情 :隐藏**名字**。至于名字相关的类型是否一致并不重要。本例中，double 类型的 x 隐藏了 int 类型的 x。

引入继承。我们知道当我们在一个派生类的成员函数中企图引用基类的某些内容（比如成员函数、typedef、或者数据成员等等）时，编译器能够找出我们所引用的内容，因为派生类所继承的内容在基类中都做过声明。这里真正的工作方式实际上是：派生类的作用域嵌套在基类的作用域中。请看下面示例：

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf2();
  void mf3();

  ...
```

```
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf4();

  ...
};
```



本示例中同时存在公共的、私有的名字，另外同时包含了数据成员和成员函数的名字。成员函数还包括纯虚函数、简单虚函数（非纯虚的）和非虚函数。这就是向大家强调，我们此处讨论的中心话题就是**名字**。示例中还可以添加类型的名字，比如枚举类型、嵌套类以及预定义类型的名字。这里讨论的核心是：它们都是名字，而它们是为哪些东西命名的并不重要。示例中使用了单一继承结构，然而一旦你了解了 C++中单一继承的行为方式之后，多重继承的行为也就不难推断了。

假定继承类中 `mf4` 是这样实现的（部分内容）：

```
void Derived::mf4()
  ...
  mf2();
  ...
}
```

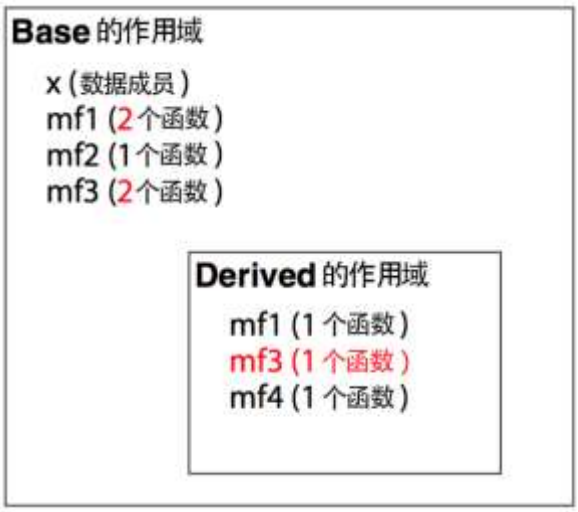
当编译器看到这个函数中使用了 mf2 这个名字，它就能够找到 mf2 的出处。编译器是这样做到的：它通过搜寻名字为 mf2 的那处声明所在的作用域。首先它在本地作用域（也就是 mf4 以内）查找，但是没有找到任何名字为 mf2 的声明。随后编译器搜寻当前包含它的域，也就是 Derived 类的作用域。仍然没有找到，于是又转向搜索上一层作用域，也就是基类。在这里编译器终于找到了名叫 mf2 的东西，于是搜索结束。如果 Base 类中依然没有 mf2，那么搜索仍会继续，从包含 Base 的名字空间开始，到全局作用域为止。

虽然我刚刚描述的查找过程是精确的，但是其对于 C++中名字查找机制的描述依然没有做到面面俱到。所幸我们的目标并不是对名字查找机制刨根问底从而去编写一个编译器。我们的目标是避免恼人的意外发生，针对这一点，我们掌握的信息已经足够了。

请再次考虑上面的示例，这次我们做一些小的改动：为 mf1 和 mf3 个添加一个重载版本，并且在 Derived 中为 mf3 添加一个新版本。（如条目 36 所讲，Derived 中重载版本的 mf3（一个继承而来的非虚函数）将会使这样的设计存在无法避免的潜在危险，但是在此问题的焦点是继承下名字的可见性，我们暂且忽略这一问题。）

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  virtual void mf2();

  void mf3();
  void mf3(double);
  ...
};

class Derived: public Base {
public:
```

```
  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```



这段代码的行为将会使每个乍看到它的 C++程序员吃上一惊。由于基于作用域的名字隐藏机制并没有改变，因此基类中**所有**名叫 mf1 和 mf3 的函数都被派生类中的 mf1 和 mf3 所隐藏。从名字查找的角度看，Base::mf1 和 Base::mf3 不再被 Derived 继承！

```
Derived d;
int x;

...
d.mf1();                // 正确，调用 Derived::mf1
d.mf1(x);               // 错误！ Derived::mf1 隐藏了 Base::mf1
d.mf2();                // 正确，调用 Base::mf2

d.mf3();                // 正确，调用 Derived::mf3
d.mf3(x);               // 错误！Derived::mf3 隐藏了 Base::mf3
```

就像你所看到的，即使同一函数在基类和派生类中的参数表不同，基类中该函数依然会被隐藏，而且这一结论不会因函数是否为虚函数而改变。在本条目最开端的示例中，someFunc 中的 double x 隐藏了全局的 int x，此处的情况类似，Derived 类中的函数 mf3 也会将 Base 类中名叫 mf3 但类型不同的函数隐藏起来。

C++这一特性的理论基础是：可以防止一类继承意外的发生，那就是当你为一个库或应用框架创建一个新的派生类时，你可能会去继承远族基类中的重载版本。遗憾的是，我们通常情况下恰恰**希望**这么做。事实上，如果你使用公共继承，但不继承重载的元素，那么就有悖于公共继承的一项基本原则——基类和派生类之间是“Derived 是一个 Base”关系（见条目 32）。既然如此，你就需要时时刻刻重载C++默认情况下隐藏的继承而来的名字。

这一工作通过使用 **using 声明**来实现：

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  virtual void mf2();

  void mf3();
  void mf3(double);
  ...
};

class Derived: public Base {
public:
  using Base::mf1;       // 让基类中所有名为 mf1 和 mf3 的东西
  using Base::mf3;       // 在 Derived 的作用域中可见（并且是公有的）

  virtual void mf1();
  void mf3();
```

```
  void mf4();
  ...
};
```

现在，继承将按部就班进行：

```
Derived d;
int x;

...

d.mf1();                          // 依然正常，依然调用 Derived::mf1
d.mf1(x);                         // 现在可以了，调用了 Base::mf1

d.mf2();                          // 依然正常，依然调用 Derived::mf1

d.mf3();                          // 正常，调用 Derived::mf3
d.mf3(x);                         // 现在可以了，调用了 Base::mf3
                                  // （此处的 x 由 int 隐式转换为 double，
                                  // 从而使 Base::mf3 的调用合法。）
```

这意味着如果你继承一个包含重载函数的基类，并且你仅期望对其中一部分进行重定义或重载，你就应该为每一个不期望被隐藏的名字添加一条 `using` 声明。如果你不这样做，一些你希望继承下来的名字将可能被隐藏。

不难想象，某些场合你可能不想把基类中所有的函数继承下来。但是在公共继承体系下这是无论如何不可行的，再次声明，这是违背公共继承“`Derived`是一个`Base`”

关系的。（这也是为什么上文中 using 声明要置于派生类中的公共元素部分：因为基类中公有的名字在公共派生类中必须是公有的。）然而在私有继承体系下（参见条目 39），这种不完全继承在某些情况下是有意义的。比如，假设 Derived 类私有继承自 Base，并且 Derived 只希望继承 mf1 不包含参数的那个版本。using 声明在此就不会奏效了，因为它将使该名字所代表的**所有**继承版本的函数在派生类中可见。在这种情况下可以使用另一种技术，我们称之为"转发函数"：

```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  ...                                  // 同上
};

class Derived: private Base {
public:
  virtual void mf1()                   // 转发函数；
  { Base::mf1(); }                     // 隐式内联（参见条目 30）
  ...                                  //  （关于对纯虚函数的调用，请参见条目 34）
};

...

Derived d;
int x;

d.mf1();                               // 正常，调用 Derived::mf1
d.mf1(x);                              // 错误！Base::mf1()被隐藏了
```

内联转发函数的另一个用途是：在使用古老的编译器时，它们通常不支持使用 using 声明来为继承类的作用域引入继承的名字（这实际上是编译器的缺陷）。此时可以使用内联转发函数。

以上是继承和名字隐藏的全部内容，但是如果继承涉及模板，那么我们将以另一种形式面对“继承而来的名字被隐藏”这一问题。对于模板继承的全部细节，请参见条目 43。

时刻牢记

- 派生类中的名字会将基类中的名字隐藏起来。在公共继承体系下，这是我们永远不希望见到的。
- 为了让被隐藏名字再次可见，可以使用 `using` 声明或者转发函数。

条目34： 区分清接口继承和实现继承

公共继承的概念看似简单，似乎很轻易就浮出水面，然而仔细审度之后，我们会发现公共继承的概念实际上包含两个相互独立的部分 :函数接口的继承和函数实现的继承。二者之间的差别恰与函数声明和函数实现之间相异之处(本书引言中有介绍)等价。

假如你是一个类设计人员，某些场合下你需要使派生类仅仅继承基类成员函数的接口（声明）。而另一些时候你需要让派生类继承将函数的接口和实现都继承过来，但还期望可以覆盖继承而来的具体实现。另外，你还可能会希望在派生类中继承函数的接口和实现，而不允许覆盖任何内容。

为了获取上述三种选项的直观感受，可以参考下面的类层次结构实例，该实例用于在图形程序中表示几何形状：

```
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const std::string& msg);
  int objectID() const;
  ...
};

class Rectangle: public Shape { ... };
```

```
class Ellipse: public Shape { ... };
```

Shape 是一个抽象类，纯虚函数 draw 标示着这一点。因此客户便无法创建 Shape 类的实例，只能由 Shape 类继承出新的派生类。不过，Shape 对所有由它（公共）继承出的类有着较强的影响，因为

- 成员函数的接口总会被继承下来。就像条目 32 中所解释的，公共继承意味着“A 是一个 B”的关系，因此对于基类成立的任何东西，对于派生类也应成立。由此可知，如果一个函数对某个类适用，那么它同样也适用于这个类的派生类。

Shape 类中声名了三个函数，第一个是 draw，用于把当前对象绘制在一个假想的显示设备上。第二个是 error，在成员函数需要报告错误时它将被调用。第三个是 objectID，为当前对象返回一个标识身份的整数值。每个函数的声明方式各不相同：draw 是一个纯虚函数，error 是一个简单（非纯虚的）虚函数，objectID 是一个非虚函数。那么这些不同的声明方式的具体实现又是什么样的呢？

首先请看纯需函数 draw：

```
class Shape {
public:
  virtual void draw() const = 0;
  ...
};
```

纯虚函数最为显著的两个特征是：首先，在所有派生出的实体类中，必须要对它们进行重新声明；其次，纯虚函数在抽象类中一般没有定义内容。融合以上两点我们可以看出：

- 声明纯虚函数的目的就是让派生类仅仅继承函数接口。

对于 Shape::draw 函数来说上面的分析再恰当不过了，因为“所有的 Shape 对象必须能够绘制出来”这一要求十分合理，但是“由 Shape 类来提供缺省的具体实现”就显得很牵强了。比如说，绘制一个椭圆的算法与绘制一个长方形大相径庭。Shape::draw 告诉具体派生类的设计者：“你必须要提供一个 draw 函数，但是我可不知道你要怎么去实现它。”

顺便说一句，为纯虚函数提供一个定义并没有被 C++所禁止。也就是说，你可以为 Shape::draw 提供一套具体实现，而 C++不会报错，但是在调用这种函数时，必须要加上类名：

```
Shape *ps = new Shape;              // 错！Shape 是抽象类

Shape *ps1 = new Rectangle;         // 正确
ps1->draw();                        // 调用 Rectangle::draw

Shape *ps2 = new Ellipse;           // 正确
ps2->draw();                        // 调用 Ellipse::draw

ps1->Shape::draw();                 // 调用 Shape::draw

ps2->Shape::draw();                 // 调用 Shape::draw
```

上文所述的这一 C++特征，除了作为你在鸡尾酒会上的谈资以外，似乎真正的用处很有限。然而就像你在下文中见到的一样，这一特征也有一定的用武之地，它可以为简单（非纯）虚函数提供“超常安全”的默认具体实现。

简单虚函数背后隐藏的内情与纯虚函数有些许不同。一般情况下，派生类继承函数接口，但是简单虚函数提供了一个具体实现，派生类中可以覆盖这一实现。如果你稍加思索，你就会发现：

- 声明简单虚函数的目的就是：让派生类继承函数接口的同时，继承一个默认的具体实现。

请观察以下情形中的 Shape::error：

```
class Shape {
public:
  virtual void error(const std::string& msg);
  ...
};
```

接口要求每个类必须要提供一个 error 函数，以便在程序出错时调用，但是每个类都有适合自己的处理错误的方法。如果一个类并不想提供特殊的错误处理机制，

那么它就可以返回调用 Shape 中提供的默认机制。也就是说，Shape::error 的声明就是告诉派生类的设计者，“你应该提供一个 error 函数，但是如果你不想自己编写，那么也可以借助于 Shape 类的默认版本。”

实践表明，允许简单虚函数同时提供函数接口和默认实现是不安全的。至于原因，你可以设想一个 XYZ 航空公司的航班层次结构。XYZ 只有两种飞机：A 型和 B 型，它们飞行的航线是完全一致的。于是，XYZ 这样设计了层次结构：

```
class Airport { ... };              // 表示飞机场

class Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
};

void Airplane::fly(const Airport& destination)
{
  默认代码：使飞机抵达给定的目的地
}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };
```

此处 Airplane::fly 声明为虚函数，这是为了表明所有飞机必须要提供一个 fly 函数，同时也基于以下事实：理论上讲，不同型号的飞机需要提供不同版本的 fly 函数实现。然而，为了避免在 ModelA 和 ModelB 中出现同样的代码，我们将默认的飞行行为放置在 Airplane::fly 中，由 ModelA 和 ModelB 来继承。

这是一个经典的面向对象设计方案。当两个类共享同一特征（即它们实现 fly 的方式）时，我们将这一共同特征移动到一个基类中，然后由两个派生类来继承这一共同特征。这一设计方案使得共同特征显性化，避免了代码重复，为未来的更新工作提供了便利，减轻了长期维护的负担——所有的一切都是面向对象技术极力倡导的。XYZ 航空公司应该感到十分骄傲了。

现在请设想：XYZ 公司有了新的业务拓展，他们决定引进一款新型飞机——C 型。C 型飞机在某些方面与 A 型和 B 型有着本质的区别，尤其是，C 型飞机的飞行方式与前两者完全不同。

XYZ 的程序员将 C 型飞机添加进层次结构，但是由于他们急于让新型飞机投入运营，他们忘记了重定义 `fly` 函数：

```
class ModelC: public Airplane {

  ...                                  // 没有声明任何 fly 函数
};
```

于是，在他们的代码中将会遇到下面代码中类似的问题：

```
Airport PDX(...);                      // PDX 是我家附近一个飞机场

Airplane *pa = new ModelC;

...

pa->fly(PDX);                          // 调用了 Airplane::fly！
```

这将是一场灾难：因为此处做了一项可怕的尝试，那就是让 `ModelC` 以 `ModelA` 和 `ModelB` 的形式飞行。你将为这一尝试付出惨痛的代价。

问题的症结不在于 `Airplane::fly` 使用默认的行为，而是在没有显式说明的情况下 `ModelC` 需要继承该行为的情况下就继承了它。幸运的是以下这一点我们很容易做到：根据需要为派生类提供默认行为，如果派生类没有显式说明，那么就不为其提供。做到这一点的秘诀是：切断虚函数的**接口**和默认**实现**之间的联系。以下是一种实现方法：

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...
```

```
protected:
  void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
  默认代码：使飞机抵达给定目的地
}
```

请注意这里的 `Airplane::fly` 是如何转变成一个纯虚函数的。它为飞行提供了接口。默认实现在 `Airplane` 类中也会出现，但是现在它是以一个独立函数的形式存在的——`defaultFly`。诸如 `ModelA` 和 `ModelB` 此类需要使用默认行为的类，只需要简单地在它们的 `fly` 函数中内联调用 `defaultFly` 即可（请参见条目 30 中介绍的关于内联和虚函数之间的联系）：

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...
};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...
};
```

对于 `ModelC` 类而言，继承不恰当的 `fly` 实现是根本不可能的，因为 `Airplane` 中的纯虚函数 `fly` 强制 `ModelC` 提供自己版本的 `fly`。

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);
```

```
  ...
};

void ModelC::fly(const Airport& destination)
{
  使C型飞机抵达目的地的代码
}
```

这一方案亦非天衣无缝（程序员仍然会“复制/粘贴”出新的麻烦），但是它至少要比原始的设计方案更可靠。至于 Airplane::defaultFly，由于此处它是 Airplane 及其派生类真实的实现。客户只需要关注飞机可以飞行，而无须理会飞行功能是如何实现的。

Airplane::defaultFly 是一个非虚函数，这一点同样重要。这是因为任何派生类都不应该去重定义这一函数，这是条目 36 所致力于讲述的议题。如果 defaultFly 是虚函数，那么你将会遇到一个递归的问题：如果一些派生类忘记了重定义 defaultFly，那么它会怎样呢？

类似于上文中介绍的 fly 和 defaultFly 函数，为接口和默认实现分别提供不同函数的方法，受到了一些人的质疑。他们指出，尽管他们不怀疑将接口和默认实现分开处理的必要性，但是这样做导致一些近亲函数名字，从而污染了类名字空间。那么如何解决这一看上去自相矛盾的难题呢？我们知道纯虚函数在具体的派生类中必须得到重新声明，但是纯虚函数自身也可以有具体实现，借助这一点问题便迎刃而解。下面代码中的 Airplane 层次结构就利用了“纯虚函数自身可以被定义”这一点：

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
  ...
};

void Airplane::fly(const Airport& destination)
{                                       // 纯虚函数的具体实现
  默认代码：使飞机抵达给定的目的地
}
```

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }
  ...
};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }
  ...
};

class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
};

void ModelC::fly(const Airport& destination)
{
  使C型飞机抵达目的地的代码
}
```

这一设计方案与前一个几乎是一致的。只是这里用纯虚函数 `Airplane::fly` 代替了独立函数 `Airplane::defaultFly`。从本质上讲，这里的 `fly` 被分割成了两个基本的组成部分，它的声明确定它的接口，而它的定义确定它的默认行为（派生类可以使用这一定义，但只有在现实请求的前提下才可以）。然而将 `fly` 和 `defaultFly` 合并起来，你就失去了将这两个函数置于不同保护层次的能力：原先受保护的代码（`defaultFly` 中的代码）现在是公共的了（因为这些代码移动到了 `fly` 中）。

最后，让我们把话题转向 `Shape` 中的非虚函数——`objectID`：

```
class Shape {
public:
  int objectID() const;
  ...
};
```

当一个成员函数不是虚函数时，你不应该期待它会在不同的派生类中存在不同的行为。事实上，非虚成员函数确立了一个“**个性化壁垒**”，因为它确保了无论派生类在其他部分如何进行个性化，本函数所确定的行为不能被改变。也就是说：

- 声明一个非虚函数的目的就是让派生类继承这一函数的接口，同时强制继承其固定的具体实现。

你可以把 shape::objectID 的声明想象成：每个 Shape 对象都有一个函数能生成“对象身份标识”的函数。这一“对象身份标识”总是以同一方式运行。这一方式由 Shape::objectID 的定义确定，任何派生类都不能偿试更改这一方式。因为一个非虚函数就确定了一个“个性化壁垒”，所以在任何的派生类中都不允许重定义该函数，这一点将在条目 36 中详细讲解。

纯虚函数，简单虚函数和非虚函数，不同的声明方式使你能够精确地指定你的派生类需要继承什么：是仅仅继承接口，还是同时继承接口和实现，抑或接口和一套固定的实现。由于这些不同种类的声明意味着彼此在基础层面存在着不同，因此你在声明成员函数时，一定要仔细斟酌。如果你这样做了，那么你将避免缺乏经验的类设计人员常犯的两类错误：

首先，第一类错误是：将所有函数都声明为非虚的。这样做可以说断送了派生类进行拓展的后路。非虚析构函数更是陷阱重重（参见条目 7）。当然，设计一个不需要作为基类的类无可厚非，这种情况下，清一色的一组非虚函数也是合乎情理的。然而，由于人们常常忽视虚函数和非虚函数之间的差异，还有对于“虚函数会对性能产生影响”的无端猜疑，导致我们的程序中充斥着过量的完全不包含虚函数的类。但事实上，几乎每个需要充当基类的类都需要虚函数的支持。(同样请参见条目 7)

如果你谈到虚函数的性能开销问题，请允许我引用基于经验主义的“80-20 法则”(同样参见条目 30)，在一个典型的程序中，80%的运行时间将花费在 20%的代码上。这一法则十分重要，因为它意味着在一般情况下，80%的虚函数调用将不会对你的程

序的整体性能造成任何影响。与其为虚函数是否会带来无法承受的性能开销而顾忌重重，还不如把精力放在程序中真正会带来影响的那 20%上。

另一个一般的问题是：将**所有的**成员函数都声明为虚函数。有时候这么做是正确的——条目 31 中的接口类就是证据。然而，这样做给人的印象就是这个类的设计者缺乏主心骨。在派生类中一些函数**不应该**进行重定义，你必须要通过将这些函数声明为非虚函数才能确保这一点。你应该清楚，并不是让客户去重定义所有的函数，你的类就成了万能的了。如果你的类中包含个性化壁垒，那么就应该大胆的将其声明为非虚函数。

时刻牢记

- 接口继承与实现继承存在着不同。在公共继承体系下，派生类总是继承基类的接口。
- 纯虚函数要求派生类仅继承接口。
- 简单（非纯）虚函数要求派生类在继承接口的同时继承一个默认的实现。
- 非虚函数要求派生类继承接口和强制固定内容的实现。

条目35： 虚函数的替代方案

假设你正在设计一款游戏软件，你需要为游戏中的各种角色设计一个层次结构。你游戏的剧情中充满了风险刺激，那么游戏中的角色就会经常遇到受伤或者生命值降低的情况。于是你决定为角色类提供一个成员函数：`healthValue`，这一函数通过返回一个整数值来表示角色当前的生命值。由于计算不同角色生命值的方式可能会不一样，因此不妨将 `healthValue` 声明为虚函数，这样做十分顺理成章：

```
class GameCharacter {
public:
  virtual int healthValue() const; // 返回角色的生命值
  ...                              // 派生类可以重定义该函数
};
```

此处，healthValue 函数并没有直接声明为纯虚函数，这一事实告诉我们计算生命值存在着一种默认的算法（参见条目 34）。

事实上，所谓“顺理成章”的方法，在某些情况下却会成为工作的绊脚石。由于这一设计太过“顺理成章”了，你也许就不会再为寻找替代方案花费足够的精力。为了让你从面向对象设计的思维定势中解脱出来，让我们另辟蹊径，解决这一问题。

模板方法模式：通过“非虚接口惯用法”实现

我们从讨论一个有趣的思想流派开始，这一流派坚持虚函数必定为私有函数。其追捧者也许会提出，更优秀的方案中 healthValue 仍为公共成员函数，但将其声明为非虚函数，并让其调用一个私有的虚函数，真正的工作由这个虚函数来完成。不妨将其命名为 doHealthValue：

```
class GameCharacter {
public:
  int healthValue() const              // 派生类不能对其进行重定义（参见条目 36）
  {
    ...                                // 准备工作：参见下文
    int retVal = doHealthValue();      // 进行实际的工作
    ...                                // 后期工作：参见下文
    return retVal;
  }
  ...

private:
  virtual int doHealthValue() const    // 派生类可以对其进行重定义
  {
    ...                                // 计算角色生命值的默认算法
  }
};
```

上文的代码中（以及本条目中所有其它代码中），我将成员函数的定义内容放置在了类定义中。如同条目 30 中所解释的，这样做使得这些函数隐式带有内联属性。我这样做只是为了让所说明的问题更加简单明了。是否选择内联与本条款的设计方

案无关，因此不要认为这里在类的内部编写成员函数是出于什么目的的。不要误会了。

让客户通过调用公有的非虚成员函数来间接的调用私有虚函数——这是一个基本的设计方案。我们一般将其称为“**非虚拟接口惯用法**（non-virtual interface，简写为NVI）”。这一方案是一个更为一般化的设计模式“模板方法”（可惜这一模式和 C++中的模板并没有什么关系）的一个特定的表现形式。我们将这些非虚函数（比如上文中的 `healthValue`）称为虚函数的“**包装器**”。

请留意上文代码中的“准备工作”、“后期工作”这两段注释内容。在虚函数做实际工作的前后，这两段注释处的代码肯定会得到调用。这是 NVI 惯用法的一个优势，这样做意味着在程序调用虚函数之前，相关的背景工作已经设定好了；在调用虚函数以后，后台清理工作也能得以进行，这两项工作都是由包装器确保进行的。举例说，“准备工作”可能包含互斥锁加锁、日志记录、确认当前类的约束条件和函数的先决条件是否满足，等等。“后期工作”可能包括互斥锁解锁、检查函数的后置条件、重新检查类的约束条件，等等。如果你让客户直接调用虚函数的话，那么这些工作也就很难开展了。

你也许注意到了：在使用 NVI 惯用法时，我们可以在派生类中对私有虚函数进行重定义，而这些函数在派生类中是无法调用的啊！这一点看上去匪夷所思，但实际上这里并不存在设计上的矛盾。虚函数的重定义工作确认了它实现其功能的方式，虚函数的调用工作确认了它实现其功能的时机。两者是相互独立的：NVI 惯用法允许在派生类中重定义虚函数，这样做使得基类将虚函数调用方式的选择权赋予派生类，然而基类仍保留函数调用时机的决定权。这一点乍看上去有些异样，但是，“允许派生类对私有虚函数进行重定义”这一 C++规则是非常合理的。

在 NVI 惯用法的背景下，并没有严格要求虚函数必须是私有的。在一些类层次结构中，一个虚函数可能在某个派生类中的实现版本中调用基类的其他成员（参见条目 27 中 `SpecialWindow` 的例子），为了让这样的调用合法，这个虚函数必须声明为受保护的，而不是私有的。还有一些情况下虚函数甚至要声明为公有的（参见条目 7：多态基类中的析构函数），在这些情况下 NVI 惯用法不再可用。

策略模式：通过函数指针实现

NVI 惯用法是公有虚函数的一个有趣的替代方案，但是从设计的观点来看，这样做无异于粉饰门面罢了。毕竟我们仍然在使用虚函数来计算每一个角色的生命值。这里存在着一个改进效果更为显著的设计主张：计算角色生命值的工作应与角色类型完全无关，于是计算生命值的工作便无须为角色类的成员。举例说，我们可以让每个角色类的构造函数包含一个函数指针参数，并由这一参数将生命值计算方法函数传入，我们可以通过调用这个函数进行实际的计算工作：

```
class GameCharacter;                // 前置声明

// defaultHealthCalc 函数：计算生命值的默认算法
int defaultHealthCalc(const GameCharacter& gc);

class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);

  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
  : healthFunc(hcf)
  {}

  int healthValue() const
  { return healthFunc(*this); }
  ...
private:
  HealthCalcFunc healthFunc;
};
```

这一方案是对另一个常见的设计模式——“策略模式”的一个简单应用。与使用虚函数的 GameCharacter 层次结构解决方案相比而言，本方案为我们带来了一些有趣的灵活性：

- 同一角色类型的不同实例可以使用不同的生命值计算函数。比如：

```
class EvilBadGuy: public GameCharacter {
public:
```

```
  explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc)
  : GameCharacter(hcf)
  { ... }
  ...
};
int loseHealthQuickly(const GameCharacter&); // 不同计算方式的
int loseHealthSlowly(const GameCharacter&);  // 生命值计算函数

EvilBadGuy ebg1(loseHealthQuickly);          // 同一类型的角色
EvilBadGuy ebg2(loseHealthSlowly);           // 使用不同类型的
                                             // 生命值计算方法
```

- 对特定的角色的生命值计算函数可以在运行时更改。比如说，GameCharacter 可以提供一个名为 setHealthCalculator 的成员函数，我们可以使用这个函数来更换当前的生命值计算函数。

另一方面，我们发现，生命值计算函数不再是 GameCharacter 层次结构的成员函数了，这一事实说明，这类生命值计算函数不再有访问它们所处理对象的内部成员的特权。比如说，defaultHealthCalc 函数对于 EvilBadGuy 类中的非公有成员就没有访问权限。如果角色的生命值可以单纯依靠角色类的公有接口所得到的信息来计算的话，上述的情况并不是问题，但是一旦精确的生命值计算需要调用非公有的信息，那么这一情况便成为问题了。事实上，凡是你使用类外部的等价功能（比如通过非成员非友元函数或通过其它类的非友元的成员函数）来替代类内部的功能（比如通过成员函数）时，都存在潜在的问题。本篇余下部分内容都存在这一问题，因为我们要考虑的其它设计方案都会涉及到 GameCharacter 层次结构外部函数的使用。

让非成员函数访问类内部的非公有成员只有一个解决办法，那就是降低类的封装度，这是一个一般的规则。比如说，类可以将非成员函数生命为友元，类也可以为需要隐藏的具体实现提供公有访问函数。使用函数指针代替虚函数可以带来一些优势（比如可以为每一个角色的实例提供一个生命值计算函数，可以在运行时更换计算函数），而这样做也会降低 GameCharacter 的封装度，至于优缺点之间孰轻孰重，你在实战中必须做到具体问题具体分析，尽可能的审时度势。

策略模式：通过 tr1::function 实现

如果你对模板和模板的隐式接口的用法（参见条目 41）很熟悉的话，那么上述的“基于函数指针”的方案就显得十分蹩脚了。我们考虑：旧的方案必须使用一个函数来实现生命值计算的功能，能不能用其它一些东西（比如函数对象）来代替这个函数呢？为什么这个函数不能是成员函数呢？还有，为什么这个函数一定要返回一个整数值，而不能返回一个可以转换成 int 类型的其他对象呢？

如果我们使用一个 tr1::function 类型的对象来代替函数指针，那么上述问题中的约束则会瞬间瓦解。就像条目 54 中所解释的，tr1::function 对象可以保存**任意可调用实体**（也就是：函数指针、函数对象、或成员函数指针），这些实体的签名一定与所期待内容的类型兼容。以下是我们刚刚看到的设计方案，这次加入了 tr1::function 的使用：

```
class GameCharacter;                // 同上
int defaultHealthCalc(const GameCharacter& gc);
                                    // 同上
class GameCharacter {
public:
// HealthCalcFunc 可以是任意可调用的实体，
// 它可以被任意与 GameCharacter 类型兼容的对象调用
// 且返回值必与 int 类型兼容，详情见下文
   typedef std::tr1::function<int (const GameCharacter&)>
           HealthCalcFunc;
   explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
   : healthFunc(hcf)
   {}

   int healthValue() const
   { return healthFunc(*this); }

   ...

private:
  HealthCalcFunc healthFunc;
};
```

正如你所见到的，HealthCalcFunc 是对 tr1::function 某个实例的一个 typedef。这意味着它与一般的函数指针类型并无二致。请仔细观察，HealthCalcFunc 所表示的自定义类型：

```
std::tr1::function<int(const GameCharacter&)>
```

在这里，我对此 tr1::function 实例“目标签名”做加粗处理。这里目标签名为“一个包含 const GameCharacter& 参数并返回 int 类型的函数”。此 tr1::function 类型（也就是 HealthCalcFunc 类型）的对象可以保存兼容此目标签名的任意可调用实体。“兼容”意味着实体中必须包含（或可隐式转换为）const GameCharacter&类型的参数，并且此实体必须返回一个（或可隐式转换为）int 类型的返回值。

与上文中我们刚刚看到过的设计方案(GameCharacter 保存一个函数指针)相比，本方案并没有显著的改动。二者仅有的差别就是后一版本中 GameCharacter 可以保存 tr1::function 对象——指向函数的**一般化**指针。这一改变实际上是十分微不足道的，甚至有些跑题，但是，它显著提高了客户在程序中修改角色的生命值计算函数的灵活性，这一点还是没有偏离议题的：

```
short calcHealth(const GameCharacter&);       // 生命值计算函数
                                              // 注意：返回值类型非 int

struct HealthCalculator {                     // 生命值计算类
  int operator()(const GameCharacter&) const // 函数对象
  { ... }
};

class GameLevel {
public:
  float health(const GameCharacter&) const;  // 生命值计算成员函数
  ...                                         // 注意：返回类型非 int
};

class EvilBadGuy: public GameCharacter {      // 同上
  ...
};
```

```
class EyeCandyCharacter: public GameCharacter {
  ...                                          // 另一个角色类型;
};                                             // 假设与 EvilBadGuy
                                               // 有同样的构造函数

EvilBadGuy ebg1(calcHealth);                   // 使用函数的角色

EyeCandyCharacter ecc1(HealthCalculator());    // 使用函数对象的角色

GameLevel currentLevel;
...
EvilBadGuy ebg2( std::tr1::bind(&GameLevel::health, currentLevel, _1) );
                                               // 使用成员函数的角色
                                               // 详情见下文
```

我个人认为，tr1::function 可以把你带入一个全新的美妙境界，它让我“热血沸腾”。如果你的热血暂时没有那么“沸腾”，那可能是因为你还纠结于 ebg2 的定义，你还不了解 tr1::bind 的工作机理。请允许我对它做出介绍。

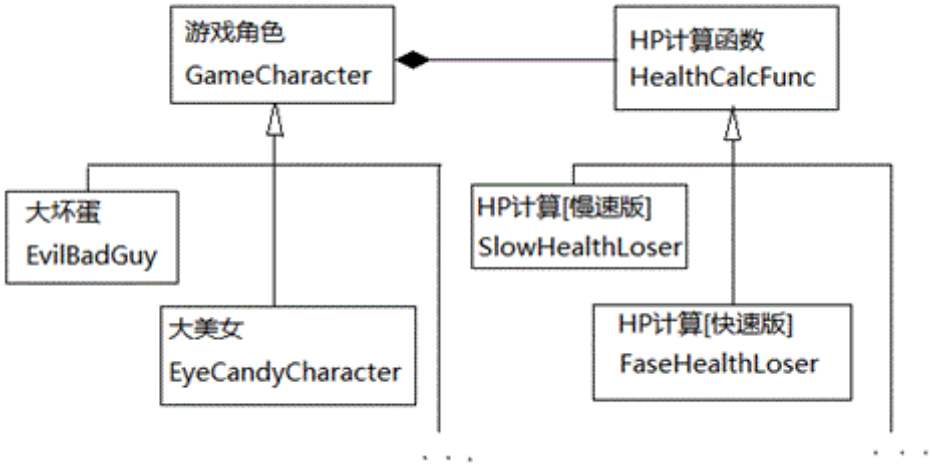
我要说的是，要计算 ebg2 的生命值，需要使用 GameLevel 类中的 health 成员函数。现在，我们所声明的 GameLevel::health 函数表面上看包含一个参数（一个指向 GameCharacter 对象的引用），但是它实际上包含两个参数，这是因为它还包含一个隐式的 GameLevel 参数（this 指针所指对象）。然而，GameCharacter 类的生命值计算函数仅包含一个参数（需要计算生命值的 GameCharacter 对象）。如果我们使用 GameLevel::health 函数来计算 ebg2 的生命值的话，为了“适应”它，我们这里只能使用一个参数（GameCharacter），而不能使用两个（GameCharacter， GameLevel），这种情况有些蹩脚。本例中，对于 ebg2 的生命值计算，我们希望始终选用 currentLevel 作为 GameLevel 对象，我们可以将 currentLevel 作为 GemeLevel 对象将两者“绑定”（bind）起来，这样，我们在每次使用 GameLevel::health 函数来计算 ebg2 的生命值时，都可以使用绑定好的二者。这就是 tr1::bind 调用所做的：他告诉 ebg2 的生命值计算函数，应该一直指定 currentLevel 作为 GameLevel 对象。

上文中我略去了大量细节内容，比如说，为什么“_1”意味着“在为 ebg2 调用 GameLevel::health 时选用 currentLevel 作为 GameLevel 对象”。这些细

节并不太难理解，而且它们与本节所讲的基本点并无关系，通过使用 tr1::function 来代替函数指针，在计算角色的生命值时，我们可以让客户使用任意可调用实体。这是再酷不过的事情了！

“经典”策略模式

如果相对于用 C++来耍酷，你更加深谙设计模式之道，那么更加传统的策略模式是这样实现的：将生命值计算函数声明为虚函数。并在此基础上创建一个独立的生命值计算层次结构。以下的 UML 图体现了这一设计方案：



如果 UML 符号还是让你一头雾水的话，我就简单点儿说吧：GameCharacter 是一个继承层次结构中的基类，EvilBadGuy 和 EyeCandyCharacter 是派生类；HealthCalcFunc 是另一个继承层次结构的基类，其下包含 SlowHealthLoser 和 FastHealthLoser 等派生类。GameCharacter 类型的每一个对象都有一个指针，这一指针指向对应的 HealthCalcFunc 层次结构中的对象。

以下是对应的代码框架：

```
class GameCharacter;                // 前置声明

class HealthCalcFunc {
public:
  ...
```

```
  virtual int calc(const GameCharacter& gc) const
  { ... }
  ...
};

HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
  explicit GameCharacter(HealthCalcFunc *phcf = &defaultHealthCalc)
  : pHealthCalc(phcf)
  {}

  int healthValue() const
  { return pHealthCalc->calc(*this);}
  ...
private:
  HealthCalcFunc *pHealthCalc;
};
```

以上的方案可以使熟悉“标准”策略模式实现方式的朋友很快的识别出来，同时，此方案还可以通过为 HealthCalcFunc 层次结构中添加一个派生类的方式，使对现有的生命值算法作出改进成为可能。

小结

本条目中最基本的建议是：在你为当前的问题设计解决方案时，不妨考虑一下虚函数以外的其他替代方案。以下是对上述替代方案的概括：

- 使用“非虚拟接口”惯用法（NVI 惯用法），它是模板方法设计模式的一种形式，它将公有非虚成员函数包装成为更低访问权限的虚函数。
- 使用函数指针数据成员代替虚函数。策略设计模式一个简装的表现形式。
- 使用 tr1::function 数据成员代替虚函数。可以使用任意可调用实体，只要实体特征与所期待的相兼容即可。同样是策略设计模式的一种形式。

- 使用另一个层次结构中的虚函数代替同一层次结构内的虚函数。这是策略设计模式的传统实现形式。

以上并不是虚函数替代方案的完整列表，但是这足以说服你虚函数**是**存在替代方案的。此外，这些方案的比较优势和劣势，清楚的告诉你在应用的过程中**应该**对它们加以考虑。

为了避免陷入面向对象设计的思维定势，我们不妨不失时机的另辟蹊径，所谓“条条大路通罗马”。花些时间来研究这些替代方案还是颇有裨益的。

时刻牢记

- 虚函数的替代方案包括：NVI 惯用法和策略模式的不同实现方式。NVI 惯用法是模板方法设计模式的一个实例。
- 将成员函数的功能挪至类外部存在着以下缺点：非成员函数对类的非公有成员没有访问权限。
- `tr1::function` 对象就像更一般化的函数指针。这类对象支持给定特征的任意可调用实体。

条目36： 避免对派生的非虚函数进行重定义

现在考虑以下的层次结构：B 是一个基类，D 是由 B 的公共继承类，B 类中定义了一个公有成员函数 mf，由于这里 mf 的参数和返回值不是讨论的重点，因此假设 mf 是无参数无返回值的函数。即：

```
class B {
public:
  void mf();
  ...
};
class D: public B { ... };
```

即使不知道 B、D、mf 的任何信息，我们声明一个 D 的对象 x：

```
D x;                                    // x 是 D 类型的对象
```

```
B *pB = &x;                          // 指向 x 的指针
pB->mf();                            // 通过指针调用 mf 函数

D *pD = &x;                          // 指向 x 的指针
pD->mf();                            // 通过指针调用 mf 函数
```

在这里，如果告诉你 pD->mf() 与 pB->mf() 将拥有不同的行为，你很可能会感到意外。因为两种情况都调用了 x 对象的成员函数 mf，两次调用使用了同一函数和同一对象，mf() 理所应当具有一致的行为。难道不是吗？

你说得没错，的确"理所应当"。但这一点无法得到保证。在特殊情况下，如果 mf 是非虚函数并且 D 类中对 mf 进行了重定义，那么问题就出现了：

```
class D: public B {
public:
  void mf();                         // 隐藏了 B::mf; 参见条目 33
  ...
};

pB->mf();                            // 调用 B::mf

pD->mf();                            // 调用 D::mf
```

此类"双面行为"的出现，究其原因，是由于诸如 B::mf 和 D::mf 这样的**非虚**函数是静态绑定的（参见条目 37）。这也就意味着：由于我们将 pB 声明为指向 B 的指针，那么通过 pB 所调用的**所有**非虚函数都将调用 B 类中的版本，即使 pB 指向一个 B 的派生类的对象也是如此，正如上文示例所示。

另一方面，由于虚函数是动态绑定的（再次参见条目 37），因此它们不会被这个问题困扰。如果 mf 是虚函数，那么无论通过 pB 还是 pD 来调用 mf 都会是对 D::mf 的调用，这是因为 pB 和 pD 实际上指向同一对象，这个对象是 D 类型的。

如果你正在编写 D 类，并且你对由 B 类继承而来的 mf 函数进行了重定义，那么 D 类将会表现出不稳定的行为。在特定情况下，任意给定的 D 对象在调用 mf 函数时

可能表现出 B 或 D 两种不同的行为，决定因素将是指向 mf 的指针的类型，与对象本身没有任何关系。引用同样会遭遇这种令人困惑的行为。

但是，上述内容仅仅是实用层面的分析，我知道，你真正需要的是对“避免对派生的非虚函数进行重定义”这一命题的理论推导。我很乐意效劳。

条目 32 解释了公共继承意味着“A 是一个 B”，条目 34 描述了为什么在类中声明一个非虚函数是对类本身设置的“个性化壁垒”。将上述理论应用到类 B、D 和非虚你函数 B::mf 上，我们可以得到：

- 对 B 生效的所有东西对 D 也生效，这是因为每一个 D 对象都是一个 B 对象。
- 继承自 B 的类必须同时继承 mf 的接口和实现，这是因为 mf 是 B 类中的非虚函数。

现在，如果在 D 中重定义了 mf，那么你的设计方案中就出现了一个矛盾。如果 D **确实**需要与 B 不同的 mf 实现方案，与此同时，如果对于所有的 B 对象（无论多么个性化的）**确实**必须使用 B 实现版本的 mf，于是我们可以很简单地推断出：并不是每个 D 都是一个 B。这种情况下，D 并非公共继承自 B。另一方面，如果 D **确实**必须是 B 的公共继承类，与此同时，如果 D **确实**需要与 B 不同的 mf 实现版本，那么 mf 对 B 的“个性化壁垒”作用就不复存在了。这种情况下，mf 应该是虚函数。最后，如果每个 D **确实**是一个 B，与此同时，如果 mf **确实**对 B 起到了“个性化壁垒”的作用，那么 D 中并不会真正需要重定义 mf，它也不应该做出这样的尝试。

无论从哪个角度讲，我们都必须无条件地禁止对派生的非虚函数进行重定义。

如果阅读本文给你一种似曾相识的感觉，那么你一定是对阅读过的条目 7 还有印象，在那里，我们解释了为什么多态基类的析构函数必须为虚函数。如果你违背了条目 7 的思想（也就是说，你在多态基类中声明了一个非虚构函数），那么你也就同时违背了本条的思想。这是因为在派生类中继承到的非虚函数（基类的析构函数）一定会被重定义。即使派生类中不声明任何析构函数也是如此，这是因为，对于一些特定的函数，即使你不自己声明它们，编译器也会自动为你生成（参见条目 5）。从本质上讲，条目 7 只不过是本条的一个特殊情况，只是因为它十分重要，我们才把它单列出一个条目。

时刻牢记

- 避免在派生类中重定义非虚函数。

条目37： 避免对函数中继承得来的默认参数值进行重定义

让我们开门见山的讨论本话题 :可以继承的函数可以分为两种 :虚拟的和非虚拟的。然而，由于重定义一个派生的非虚函数始终是一个错误（参见条目 36），因此我们可以放心地将此处的讨论范围缩小至以下情况 :继承一个含有默认参数值的**虚**函数。

此情况下，证明本条目的结论非常简单：虚函数是动态绑定的，而默认参数值是静态绑定的。

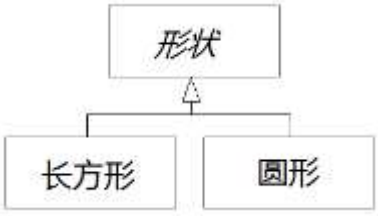
你说啥？静态绑定与动态绑定之间的区别已经让你头晕目眩了？（众所周知，静态绑定又称早期绑定，动态绑定又称晚期绑定。）我们只好复习一下了。

一个对象的**静态类型**就是你在对其进行声明时赋予它的类型。请考虑下面的类层次结构：

```
// 几何形状类
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  // 所有形状必须提供一个自我绘制函数
  virtual void draw(ShapeColor color = Red) const = 0;
  ...
};

class Rectangle: public Shape {
public:
  // 请注意：默认参数值变了——糟糕！
  virtual void draw(ShapeColor color = Green) const;
  ...
};

class Circle: public Shape {
public:
```

```
  virtual void draw(ShapeColor color) const;
  ...
};
```

用 UML 来表示：



现在请考虑下面的指针：

```
Shape *ps;                          // 静态类型 = Shape*
Shape *pc = new Circle;             // 静态类型 = Shape*
Shape *pr = new Rectangle;          // 静态类型 = Shape*
```

示例中，由于 ps，pc 以及 pr 都声明为指向 Shape 的指针，因此他们的静态类型均为 Shape*。请注意，这样做使得无论他们**实际**指向的对象是什么类型，他们的静态类型都必为 Shape*。

对象的**动态类型**是通过他当前引用的对象的类型决定的。也就是说，动态类型表明了他应具有怎样的行为。在上文的示例中，pc 的动态类型是 Circle*，pr 的动态类型是 Rectangle*。而对于 ps 来说，他在当前根本不具备动态类型，因为它（目前）还没有引用任何对象呢。

动态类型，顾名思义，在程序运行时可能会有所改变，通常是通过赋值操作发生：

```
ps = pc;                            // ps 动态类型变为 Circle*
ps = pr;                            // ps 动态类型变为 Rectangle*
```

虚函数是**动态绑定**的，这就意味着，对于一个特定的函数调用，其调用对象的动态类型将决定调用这一函数的哪个版本：

```
pc->draw(Shape::Red);               // 调用 Circle::draw(Shape::Red)
pr->draw(Shape::Red);               // 调用 Rectangle::draw(Shape::Red)
```

我知道这些都是老生常谈了，你当然已经对虚函数有了透彻的理解。只有在虚函数包含默认参数值时，情况才有所不同。这是因为（如上文所述），虚函数是动态绑定的，但是默认参数是静态绑定的。这也就意味着对于一个虚函数，你可能会调用它在**派生类**中的定义，而默认参数值则采用**基类**中的值：

```
pr->draw();                    // 调用 Rectangle::draw(Shape::Red)!
```

这种情况下，由于 pr 的动态类型是 Rectangle*，于是此处便调用了虚函数 draw 的 Rectangle 版本，正如你所愿。在 Rectangle::draw 中，默认参数值是 Green。然而，因为 pr 的静态类型是 Shape*，这里的 draw 调用将采用 Shape 类中的默认参数值，而不是 Rectangle！最终，在 Shape 类和 Rectangle 类之间，对于 draw 的调用必将出现混乱的无法预知的现象。

虽然 ps，pc 和 pr 是指针，但是并不影响上文的结论。如果它们是引用的话，问题同样存在。这里只有一个重点：draw 是虚函数，他的一个默认参数值在派生类中被重定义了。

是什么让 C++在处理这一问题时如此不合常理？答案是：运行时效率。如果默认参数值是动态绑定的话，那么编译器必须提供一整套方案，为运行时的虚函数参数确定恰当的默认值。而这样做，比起 C++当前使用的编译时决定机制而言，将会更复杂、更慢。鱼和熊掌不可兼得，C++将设计的中心倾向了速度和简洁，你在享受效率的快感的同时，如果你忽略本条目的建议，你就会陷入困惑。

一切看上去似乎尽善尽美了，但是一旦你不假思索的遵守本条建议，为基类和派生类分别提供默认参数值的话，看看将会发生什么：

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  virtual void draw(ShapeColor color = Red) const = 0;
  ...
};
class Rectangle: public Shape {
public:
  virtual void draw(ShapeColor color = Red) const;
```

```
  ...
};
```

吁……恼人的重复代码。还有更糟的：这些重复代码彼此还有依赖：如果 Shape 中的默认参数值改变了的话，那么所有的派生类中相应的值都必须改变。否则这些函数仍将改变继承来的默认参数值。那么怎么办呢？

遇到麻烦了？虚函数无法按照你预想的方式运行？这时候明智的做法是：考虑一个替代的设计方案，条目 35 中介绍了几种虚函数的替代方案。其中一种是非虚拟接口惯用法方案（NVI 惯用法）：在基类中用一个公有的非虚函数调用一个私有的虚函数，并在派生类中重定义这一虚函数。在这里，我们将默认参数置于非虚函数中，让虚函数做具体的工作。

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  void draw(ShapeColor color = Red) const
  {                                  // 现在 draw 是非虚函数
    doDraw(color);                   // 调用一个虚函数
  }
 ...

private:
  virtual void doDraw(ShapeColor color) const = 0;
                                     // 这个函数做真正的工作
};

class Rectangle: public Shape {
public:
  ...

private:
  virtual void doDraw(ShapeColor color) const;
                                     //此处不需要默认参数值
  ...
```

```
};
```

由于在派生类中不能对非虚函数进行重载（参见条目 36），因此，显然地，这一设计方案使得 `draw` 函数中 `color` 参数的默认值永远为 `Red`。

时刻牢记

- 避免在对函数中继承得来的默认参数值进行重定义，这是因为默认参数值是静态绑定的，而虚函数（派生类中唯一的一系列可以重定义的函数）是动态绑定的。

条目38： 使用组合来表示“A 拥有一个 B”、“A 以 B 的形式实现”

当一个类型 A 的对象中包含另一个类型 B 的对象时，我们说 A 与 B 之间的关系是“**组合**”。请看示例：

```
class Address { ... };              // 住址

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  std::string name;                 // 组合对象
  Address address;                  // 同上
  PhoneNumber voiceNumber;          // 同上
  PhoneNumber faxNumber;            // 同上
};
```

上述示例中，`Person` 对象由 `string`、`Address` 和 `PhoneNumber` 三种对象组合而成。在程序员之间，“组合”一词拥有众多的同义词。诸如：**分层**、**包含**、**聚合**和**嵌入**。

条目 32 中解释了公共继承意味着“A 是一个 B”。同时组合也有其内涵，事实上它拥有两个内涵，组合既可以表示“A 拥有一个 B”，也可以表示“A 以 B 的形式实现”。这是由于在你的软件中你针对的是两个不同的领域。你的程序中的一些对象与你正在建模的世界相关，比如人、车、视频帧等等。这些对象则存在于**应用领域**。而另一些对象单纯是为了程序的具体实现人为创造的，诸如缓冲区、互斥锁、搜索树，等等。这些类型的对象则针对软件中的**实现领域**。当组合出现在应用域内的对象之间时，它表达的是“A 拥有一个 B”的关系；而组合出现在实现域中，则意味着“A 以 B 的形式实现”。

上文中的 `Person` 类演示了“A 包含 B”的关系。一个人——`Person` 对象“拥有”一个姓名（`name`）、一个住址（`address`）、一个电话号码（`voiceNumber`）、一个传真号码（`faxNumber`）、你不能说“人**是**姓名”、“人**是**地址”这样的话。应该说“人**有**姓名”、“人**有**住址”。大多数人还是能够轻易区分“是”和“有”之间的区别的。因此“A 是 B”和“A 拥有 B”两者并不易混淆。

某种意义上讲，“A 是 B”与“A 以 B 的形式实现”二者之间的区别更让人难以分辨。举例说，假设你需要一个表示集合的模板，其中容纳的对象的数目非常有限。即该集合中不允许存在重复的对象。由于复用在 OOP 的世界里是十分美妙的事情，你会本能的想到使用标准库中的 `set` 模板。有现成的工具为什么不加以利用呢。

不幸的是，`set` 模板的具体实现中一般每个元素都会有三个指针的开销。这是因为 `set` 通常被实现为平衡搜索树，这种数据结构确保了查找、插入、删除操作的时间复杂度均为 $O(\lg n)$。在效率至上的环境中，这种设计方案合情合理，然而在你的程序中，空间比速度更加重要，这时标准库中的 `set` 模板就变得水土不服了。看上去你需要另起炉灶。

固然，复用的确是一件美妙的事情。作为数据结构专家的你，对于实现集合有各式各样的手段，其中之一便是使用链表。你当然也了解标准 C++库中有一个 `list` 模板，因此你可以去（复）用它。

于是，你决定开辟一个全新的模板 `Set`，由 `list` 模板继承而得。也就是说 `Set<T>` 将由 `list<T>`继承而得。在你的实现中，`Set` 对象实际上将会是一个 `list` 对象。于是你这样声明 `Set` 模板：

```
template<typename T>                    // 创建 Set：此处是复用 list 的错误做法
```

```
class Set: public std::list<T> { ... };
```

这一方按乍看上去十分完美，实际上却存有隐患。如条目 32 所讲，如果 D 是一个 B，那么对于 B 成立的一切对于 D 也成立。然而，list 对象中可以存在重复的元素，因此如果我们先后两次将 3051 这个值插入 list<int>中，这个表中将存在两个 3051 的副本。相反，Set 不应含有重复的元素，如果两次插入 3051，那么 Set<int> 中应仅存在一个该值的副本。于是"Set 是一个 list"这一说法便不成立了——对于 list 对象成立的一些结论不适用于 Set 对象。

由于这两个类之间的关系不是"A 是 B"，因此使用公共继承的方式来构造两者之间的关系便是错误的。我们可以想到 Set 对象可以"以 list 的形式实现"，以下是正确的做法：

```
template<class T>                    // 创建 Set：此处是复用 list 的正确做法
class Set {
public:
  bool member(const T& item) const;
  void insert(const T& item);
  void remove(const T& item);
  std::size_t size() const;

private:
  std::list<T> rep;                  // 代表 Set 中的数据
};
```

Set 的成员函数可以全方位的依赖 list 乃至标准库中其他部分提供的各项功能，因此实现方法是简单直接的，只要你掌握 STL 的基本使用方法即可：

```
template<typename T>
bool Set<T>::member(const T& item) const
{
  return std::find(rep.begin(), rep.end(), item) != rep.end();
}

template<typename T>
void Set<T>::insert(const T& item)
{
```

```
  if (!member(item)) rep.push_back(item);
}

template<typename T>
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =
    std::find(rep.begin(), rep.end(), item);
                                    // 此处为何使用 typename 请参见条目 42
  if (it != rep.end()) rep.erase(it);
}

template<typename T>
std::size_t Set<T>::size() const
{
  return rep.size();
}
```

这些函数足够简单，我们有理由将它们声明为内联函数，然而在你做出明确决定之前，我还是建议你去条目 30 复习一下内联的相关知识。

一些人可能会说：Set 的接口应该更加遵守条目 18 中讨论的主题："设计接口要易于使用而不易误用"，是否应该让 Set 遵守 STL 容器的标准，但是这里遵守这些标准需要为 Set 添加一大批内容，这样做会淹没它与 list 之间的关系。由于本章节讨论的中心是这一关系问题，因此这里我牺牲了 STL 的兼容性，而更多考虑了讲述的清晰程度。另外，Set 接口的不完善并不会掩盖此处关于它的无须争辩的事实：其与 list 之间的关系并不是"A 是 B"（尽管乍看上去很像），而是"A 以 B 的形式实现"。

时刻牢记

- 组合与公共继承之间存在着本质区别。
- 组合在应用领域意味着"A 是 B"，在实现领域意味着"A 以 B 的形式实现"。

条目39： 审慎使用私有继承

条目 32 向我们展示了以下结论：C++将公共继承处理为“A 是一个 B”关系。比如我们给定一个 Student 类继承自 Person 类的层次结构，那么编译器则会在特定的时刻将 Student 对象隐式的转变为 Person 对象，以便使特定的函数得以成功调用，这一点 C++本身已经为我们考虑周全了。这里我们不妨继续花一点时间研究一下，在上述示例中如果使用私有继承代替公共继承会发生什么：

```
class Person { ... };
class Student: private Person { ... };
                                    // 现在是私有继承
void eat(const Person& p);          // 每个人都能吃东西

void study(const Student& s);       // 只有学生会学习

Person p;                           // p 是 Person 对象
Student s;                          // s 是 Student 对象

eat(p);                             // 正确，p 是 Person 对象

eat(s);                             // 错误！Student 对象
                                    // 不是 Person 对象
```

很明显，私有继承并不呈现“A 是一个 B”的关系。那么私有继承表示的是什么关系呢？

“哇。。。”你说，“在我们讨论私有继承的含义之前，让我们先了解一下它的行为，私有继承拥有怎样的行为呢？”好的，正如上文的代码中我们看到的，私有继承的第一条守则是：如果类之间的层次关系是私有继承的话，那么编译器一般不会将派生类对象（比如 Student）直接转换为一个基类对象（比如 Person）。这一点是与公共继承背道而驰的。这也就说明了为什么对 s 对象调用 eat 函数时会发生错误。第二条守则是：派生类中继承自私有基类的成员也将成为私有成员，即使他们在基类中用 public 或 protected 修饰也是如此。

行为就介绍到这。下面我们来讨论含义。私有继承意味着“A 以 B 的形式实现”。如果有人编写了一个 D 类私有继承自 B 类，那么他这样做的真实目的应该是想借用 B 类中某些功能或特征，而不是 B 和 D 之间在概念层面的什么关系。综上，我们说私有继承是一个纯粹的实现域的技术。（也就是为什么在派生类中继承自私有基类的一切都是私有的：这一切都是具体实现的细节）私有继承（条目 34 中引入的概念）意味着只有实现应该被继承下来，而接口则应该忽略掉。如果 D 私有继承自 B，那么这就意味着 D 对象以 B 对象的形式实现，而且再没有其他任何意义。私有继承只存在于软件**实现**的过程中，而在软件**设计**过程中永远不会涉及。

私有继承意味着“A 以 B 的形式实现”的关系，这一事实显得有些令人困惑，因为条目 38 中指出，组合可以做同样的事情。那么在这两者中要怎样做出权衡取舍呢？答案很简单：尽量使用组合，在不得已时才使用私有继承。那么什么时候才是“不得已”的情形？主要是设计中出现了受保护的成员和/或虚函数的情形，这里还存在一种边缘情形，当存在存储空间问题的情形下，我们还需要考虑更多。这个问题我们稍后再讨论，毕竟它仅仅存在于极端条件下。

现在我们来考虑一个包含 Widget 的应用程序。我们的设计要求我们对 Widget 的使用方式有一个更全面的了解。比如说，我们不仅仅需要了解诸如“Widget 成员函数调用的频率”这类问题，还需要知道“调用的频率随时间的推移有何变化”。对于拥有不同运行阶段的程序而言，在各阶段都需要有相应的行为配置与之配套。比如说，对于一个编译器而言，语法分析阶段所运行的函数，与优化和代码生成阶段的函数是大相径庭的。

我们决定修改 Widget 类以便跟踪每个成员函数被调用的次数。在运行时，我们将周期性的检查这一信息，这一工作可能与监视每个 Widget 对象的值，以及其他一切我们认为有用的数据同时进行。为了达到这一目的，我们需要设置一个某种类型的计时器，以便让我们掌握收集统计信息的时机。

复用现有代码肯定比重写新的代码更好，因此我们“翻箱倒柜”寻找出了最适合的类，下边是代码：

```
class Timer {
public:
  explicit Timer(int tickFrequency);
```

```
  virtual void onTick() const;   // 表针跳一下，自动调用一次
  ...
};
```

这正是我们需要的。对于 Timer 对象，我们可以依照需要任意设定表针跳动的频率，而且对于每一次跳动，该对象都会自动调用一个虚函数。我们可以对这一虚函数进行重定义，以使它具备监视所有 Widget 对象当前状态的功能。堪称完美！

为了让 Widget 重定义 Timer 中的虚函数，Widget 必须继承自 Timer。但是公共继承在此时就不合时宜了。我们不能说“Widget 是一个 Timer”。由于 onTick 不是 Widget 概念层面接口的一部分，因此 Widget 的客户不应该拥有调用 onTick 的权限。如果我们让客户能够调用这类函数的话，那么他们很容易就会用错 Widget 的接口。这显然违背了条目 18 中的建议：“让接口更易使用，而不易被误用”。公共继承在这里并不是一个可选方案。

于是我们使用私有继承：

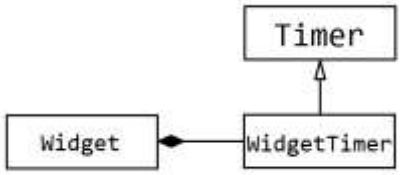
```
class Widget: private Timer {
private:
  virtual void onTick() const;    // 监视 Widget 对象的使用数据等等
  ...
};
```

借助于私有继承的美妙特性，Timer 类的公共函数 onTick 在 Widget 中变成了私有的，同时我们确保 onTick 在 Widget 中被重定义。再次强调，将 onTick 置于公共接口下将会使客户误认为这些函数可以调用，这将会违背条目 18。

这是一个不错的设计方案，但是这里使用私有继承并不是必须的，这样做并没有实际的意义。如果我们使用组合来代替也没有问题。我们可以在 Widget 类的内部声明一个私有的嵌套类，并由这个嵌套类公共继承 Timer，在此嵌套类中重定义 onTick，然后在 Widget 类中放置一个改嵌套类的对象。以下代码是这一方法的概要：

```
class Widget {
private:
  class WidgetTimer: public Timer {
  public:
```

```
    virtual void onTick() const;
    ...
  };
    WidgetTimer timer;
    ...

};
```



这一设计比仅使用私有继承的版本更加复杂，这是因为此设计中包含了公共继承和组合两种技术，并且引入了一个新的类（WidgetTimer）。坦白说，我介绍这一方案的主要目的实际上是要告诉大家设计不要局限于某种单一的方案，刻意训练自己对于同一个问题举一反三，会令你受益非浅（参见条目 35）。最后，我还是要提出两个理由来说明：公共继承加组合的方案要优于私有继承。

首先，你可能期望将 Widget 设计为可派生的类，但是同时你也希望在派生类中阻止对 onTick 的重定义。如果让 Widget 继承自 Timer，那么即便你使用私有继承，你的期望也无法完全达成。（请回忆条目 35：尽管派生类中无法调用基类中的虚函数，派生类中也可以对虚函数做出重定义。）但是，如果由 WidgetTimer 继承自 Timer，并将其放置在 Widget 中作为私有嵌套类，这样 Widget 的派生类便失去了对 WidgetTimer 的访问权限，于是 Widget 的派生类则无法继承 WidgetTimer，也无法重定义 WidgetTimer 内的虚函数。如果你曾经使用 Java 或 C#编程，你会发现这两种语言都有防止派生类重定义虚函数（Java 中的 final 方法，C#中的 sealed 方法）的功能，现在你已经了解如何在 C++中等效的实现这一行为。

其次，你可能期望使 Widget 的编译依赖程度最小化。如果 Widget 继承自 Timer，那么在编译 Widget 类时，Timer 必须有可用的定义，因此定义 Widget 类的文件可能需要添加 #include "Timer.h" 指令。另外，如果将 WidgetTimer 移

出 Widget， Widget 中仅包含一个指向 WidgetTimer 的指针，Widget 可以通过 WidgetTimer 类的简单声明来调用这一指针。这样不需要任何 #include 指令就可以使用 Timer。对于大型系统来说，这样的剥离工作可能是非常重要的。（对于最小化编译依赖议题，请参见条目 31）

上文曾强调过，私有继承主要应用于以下情形：一个“准派生类”需要访问“准基类”中受保护的部分，或者需要重定义一个或多个虚函数，但是两个类之间在概念层面的关系是“A 以 B 的形式实现”，而不是“A 是一个 B”。然而，我们也说过，当涉及到空间优化问题时，这里存在一种边缘情形，我们更倾向于使用虚拟继承，而不是组合。

此边缘情形中的“边缘”实际上是十分“锋利”的，也就是说它发生的几率很小：只有在你使用一个不存在数据的类时才会遇到。这样的类中，没有非静态数据成员，没有虚函数（因为虚函数的存在会使每一个对象中添加一个 vptr，参见条目 7），没有虚拟基类（因为虚拟基类会带来额外的体积开销，参见条目 40）。在概念层面，这些**空类**的对象不应该使用任何空间，因为每个单独的对象是没有任何数据需要存储的。然而，C++强制要求这些独立的对象必须占用一定的空间，是有其技术上的原因的。比如你写下了下面的代码：

```
class Empty {};                     // 由于没有任何数据，
                                    // 因此对象也应不占任何内存
class HoldsAnInt {                  // 应仅占一个 int 的空间
private:
  int x;
  Empty e;                          // 不应消耗任何内存
};
```

你将发现：sizeof(HoldsAnInt) > sizeof(int)。Empty 数据成员也需要内存。对于大多数编译器而言，sizeof(Empty)的值是 1，这是因为 C++中禁止存在零空间的独立对象，这一“禁令”一般是静默地通过在“空”对象中添加一个 char 成员来达成的。然而，C++对内存对齐的要求（参见条目 50）会使得编译器在诸如 HoldsAnInt 这样的类中做适当的填充，因此 HoldsAnInt 对象中很可能不仅添加了一个 char 的空间，这些对象可能被扩容到能容纳另一个 int。（我所测试的所有编译器都恰好是这样的情形）

但是你可能注意到刚才的讨论我很小心的使用了一个字眼——“独立”，这类对象不能为零空间。对于拥有派生类对象的基类就没有强制约束了，这是因为这类对象不是独立的。如果 Empty 不是包含于 HoldsAnInt 中，而是被其继承：

```
class HoldsAnInt: private Empty {
private:
  int x;
};
```

你一定会发现，sizeof(HoldsAnInt) == sizeof(int)。这一情形被称为“空基类优化（empty base optimization，简称 EBO）”，我所测试的所有编译器均支持这一特性。如果你是一个类库开发人员，你的客户更关心内存空间问题，那么 EBO 则很值得你去了解。同时还有一件事，EBO 一般只在单一层次环境中可行。一般情况下，C++对象的排列守则要求 EBO 不能适用于继承自多个基类的派生类。

在实际环境中，尽管“空”类中不包含任何非静态数据成员，但实际上它们并不真是空的。这些空类中通常会包含 typedef、枚举类型成员、静态数据成员，或非虚函数。STL 中包含很多技术层面的空类，这些类包含了诸多有用的成员（通常是 typedef），包括 unary_function 和 binary_function 这些基类，一般一些用户自定义的函数对象可以继承它们。感谢 EBO 的广泛应用，这样的继承操作一般不会为派生类带来额外的空间开销。

让我们重温基础的部分。大多数类不是空的，因此仅在极少数情况下，EBO 可以作为私有继承的合理理由。另外，大多数继承结构呈现出“A 是一个 B”关系，其应由公共继承司职，而不是私有继承。组合和私有继承都意味着“A 以 B 的形式实现”关系，但是组合更易于理解，因此你应该尽可能的使用它。

当你正在处理的两个类没有呈现“A 是一个 B”关系，并且其中一个类需要访问另一个类中受保护的成员，或者需要重定义其中一个或若干个虚函数的情况下，私有继承最有可能成为一个合理的设计策略。即使在这种情况下，我们看到配合使用公共继承和组合的方法可以得到等效的行为，只是在设计上略显复杂。当你需要描述这样的两个类之间的关系时，**审慎**使用私有继承，意味着在你考虑过其他所有的可行方案后，并且确定没有比它更合适的，才选择使用。

时刻牢记

- 私有继承意味着“A 以 B 的形式实现”。通常它的优先级要低于组合，但是当派生类需要访问基类中受保护的成员，或者需要重定义派生的虚函数时，私有继承还是有其存在的合理性的。
- 与组合不同，私有继承可以启用“空基类优化”特性。对于类库开发人员而言，私有继承对于降低对象尺寸来说至关重要。

条目40： 审慎使用多重继承

这一条目我们讨论多重继承（Multiple inheritance，简称 MI）的问题。C++社区在这一话题大致可分为两个阵营。其中一个阵营坚信：如果单一继承（SI）很不错，那么多重继承一定会更好。而另一个阵营则认为：单继承已经很优秀了，多重继承带来的麻烦远远要比优点多。本条目我们的主要目的，就是了解关于 MI 问题的上述两个观点。

当我们在设计层面讨论 MI 时，我们会遇到一系列基本的问题，其中之一就是：可能从不同的两个基类中继承得到重名的东西（比如函数，typedef，等等。）。这样就会有产生歧义的可能。举例说明：

```
class BorrowableItem {              // 可以从图书馆借到的东西
public:
  void checkOut();                  // 在图书馆对当前物品办理租借手续

  ...
};

class ElectronicGadget {
private:
  bool checkOut() const;            // 自检，返回检测是否成功

  ...
};
```

```
class MP3Player:                    // 请注意此处的 MI
  public BorrowableItem,            // (一些图书馆有 MP3 播放器出租)
  public ElectronicGadget
{ ... };                            // 类定义不重要

MP3Player mp;

mp.checkOut();                      // 产生歧义! 要调用哪个 checkOut?
```

请注意这个示例，即使这两个 checkOut 函数中只有一个是可访问的（checkOut 在 BorrowableItem 中是公共的，而 ElectronicGadget 中是私有的），这里对 checkOut 的调用也是存在歧义的。这一点与 C++在解决重载函数的调用时所遵守的规则是相符的：先找到最匹配的函数后检查该函数的可访问性。在这种情况下，由于在名字查找进程中 checkOut 是存在歧义的，因此这里即不存在重载解析，也不存在最佳匹配。于是 ElectronicGadget::checkOut 的可访问性就永远不会得到检查。

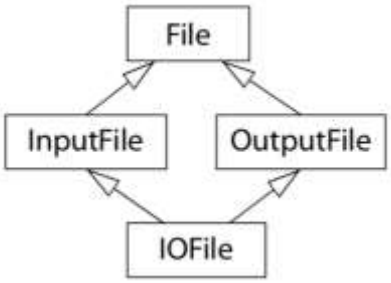
为了消除这一歧义，你必须说明调用的是哪个基类的函数：

```
mp.BorrowableItem::checkOut();      // 啊，原来是这个 checkOut...
```

当然，对于 ElectronicGadget::checkOut 你也应该采用显式调用，于是，这里的“歧义错误”就被“尝试调用一个私有成员函数错误”代替了。

多重继承就意味着一个类继承自两个及以上的基类，但是在存在 MI 的层次结构里，多重继承的若干基类还存在基类的情况也并不罕见。于是这也就会带来所谓的“MI 死亡菱形”问题。

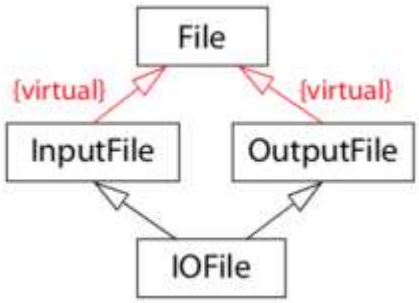
```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```



每当你遇到上面的继承层次结构中的情形：一个基类和一个派生类之间存在不同的继承路径（比如 File 和 IOFile 之间，存在 InputFile 和 OutputFile 两条继承路径）时，你必须面对这样一个问题：对于路径上的每一个基类，你是否希望其数据成员复制到派生类里。比如，假设 File 类有一个数据成员 fileName，那么 IOFile 中对 fileName 应该保留几份拷贝呢？一方面，IOFile 是由两个基类继承而来，因此这意味着它应该有两个 fileName 数据成员。另一方面，简单至上的逻辑告诉我们，一个 IOFile 应该只有一个文件名，因此即使 fileName 数据成员继承自不同的两个基类，在派生类中也不应该重复存在。

C++对这一问题的态度是搁置争议。它对两种意见都能"欣然接受"。然而在默认情况下它会保留这一重复。如果你不希望这样，你必须将保存数据的基类（对应上文中的 File 类）声明为**虚基类**。只需对直接继承自该基类的派生类使用**虚拟继承**，即可保证这一点。

```
class File { ... };
class InputFile: virtual public File { ... };
class OutputFile: virtual public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```



C++标准库中包含一个 MI 层次结构，与上文中介绍的结构很像，只是标准库中的类是类模板，而名称也不再是：`File`、`InputFile`、`OutputFile` 和 `IOFile`，取而代之的是：`basic_ios`、`basic_istream`、`basic_ostream` 以及 `basic_iostream`。

从取得正确行为的角度而言，公共继承应该时时刻刻都是虚拟的。如果这是唯一的观点，那么这里的规则就显得十分简单：只要你使用公共继承，那么就应该使用虚拟公共继承。然而，我们看这个问题不应仅仅从正确性的角度出发。在派生的对象中避免重复是需要编译器在幕后施展它的戏法的，我们得知，使用虚拟继承的类所创建的对象往往要比非虚拟继承的大一些。不同的编译器结果会略有不同，但有一点是毋庸置疑的：虚拟继承会带来额外的开销。

虚拟继承带来的开销还不止这些。比起非虚基类而言，在初始化虚拟基类时，你会面临更加复杂的规则，同时使用方式也没有前者那么直观。初始化虚拟基类的任务是由层次结构中**最后派生出的类**承担的。这一规则可能产生的后果有：（1）需要初始化的继承自虚拟基类的派生类必须时刻考虑虚拟基类的因素，不管这个派生类与基类的距离有多远。同时，（2）当在层次结构中添加一个新的派生类时，这个新的派生类也必须承担它的虚拟基类（无论是直接的还是间接的）的初始化工作。

对于虚拟基类我的建议很简单（即虚拟继承）。首先，除非不得已的情况，尽量不要使用虚拟基类。在默认情况下，请使用非虚拟继承。其次，如果某些情况下你必须使用虚拟基类，那么请不要在它们内部添加数据成员。这样你就不用再担心这些类的那些古怪的初始化（以及由此产生的赋值操作）规则了。这里有必要注意一下 Java 和.NET 里“接口”的概念。接口在很多层面上与 C++的虚拟基类都有相似之处，它的内部是不允许存在数据的。

让我们转到下一个示例，以下是一个用于为人类建模 C++的接口类：（另请参见条目 31）

```
class IPerson {
public:
  virtual ~IPerson();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};
```

IPerson 的客户必须基于 IPerson 的指针或引用进行编程，这是因为抽象类是不能够实例化的。为了创建可操作的类 IPerson 对象，IPerson 的客户可以使用工厂函数（同样请参见条目 31）来实例化一个继承自 IPerson 的具体类。

```
// 工厂函数：使用一个特定的数据库 ID 来创建一个 Person 对象；
// 至于返回类型为什么不是原始指针，请参见条目 18
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);

// 用于从客户端取得数据库 ID 的函数
DatabaseID askUserForDatabaseID();

DatabaseID id(askUserForDatabaseID());
std::tr1::shared_ptr<IPerson> pp(makePerson(id)); // 创建一个支持 IPerson
                                                  // 接口的对象
...
                                                  // 通过 IPerson 的成员函数
                                                  // 来操作*pp
```

makePerson 创建了 IPerson 对象，并以指针形式返回，它是怎么做到的呢？很显然地，此处必定存在某个继承自 IPerson 的具体类，且 makePerson 必定能够实例化这个类。

假定这个类为 CPerson，作为一个具体类，CPerson 必须为由 IPerson 继承来的纯虚函数提供具体实现。我们可以从零开始，但是若现有组件可以完成全部或绝

大多数工作，那么利用现有组件的方案就更为优秀。比如，假设有一个老版本的专为数据库设计的 PersonInfo 类，该类提供了 CPerson 所需的主要功能：

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  ...
private:
  virtual const char * valueDelimOpen() const;      // 参见
  virtual const char * valueDelimClose() const;     // 下文
  ...
};
```

你一定可以分辨出这是一个老式的类，这是因为该类的成员函数返回类型是 const char *而不是 string。但是，这个类的成员函数的名字中可以推测结果应该是十分舒服的，是的，十分舒服，不用顾忌，放心使用吧。

你会发现，PersonInfo 这样设计是为了便于使用不同的格式来打印数据库的字段，它为每个字段值在起始处和终止处各添加一个分界符号。默认是一对方括号，因此字段值“环尾狐猴”将以下面的格式呈现：

```
[环尾狐猴]
```

如果你意识到了：并不是所有的 PersonInfo 客户都希望使用方括号的方案，这里的两个虚函数——valueDelimOpen 和 valueDelimClose 将允许派生类自定义起始处和终止处的分界符号。PersonInfo 成员函数的具体实现将调用这两个虚函数来为返回值添加恰当的分界符号。以 PersonInfo::theName 为例，代码为：

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                          // 默认的起始端边界符号
}
```

```
const char * PersonInfo::valueDelimClose() const
{
  return "]";                          // 默认的终止端边界符号
}

const char * PersonInfo::theName() const
{
  // 为返回值预留缓冲区；由于 value 是静态的，因此数组中所有 char 均自动初始化为 0
  // Max_Formatted_Field_Value_Length = 格式化字段值最大长度
  static char value[Max_Formatted_Field_Value_Length];

  // 写入起始端边界符号
  std::strcpy(value, valueDelimOpen());

  将此对象的名字字段添加到 value 字符串中。(请小心避免缓冲区溢出！)

  // 写入终止端边界符号
  std::strcat(value, valueDelimClose());
  return value;
}
```

某些读者可能会质疑此处 PersonInfo::theName 过于陈旧的设计方案（尤其是这里使用了固定大小的静态缓冲区，这将使程序中存在无尽的溢出和线程问题。参见条目 21），但是请把这些问题暂时放在一边，让我们把问题的焦点置于此处："theName 中保存一个字符串用以返回计算结果。theName 首先调用了 valueDelimOpen 为字符串生成起始端边界符号，然后生成 theName 所在对象的名字，最后调用 valueDelimClose。"

由于 valueDelimOpen 和 valueDelimClose 为虚函数，因此 theName 的返回结果即取决于 PersonInfo 本身，也取决于 PersonInfo 的派生类。

对于 CPerson 的实现者来说这是一个好消息，这是因为当你精读 IPerson 的文档时你会发现存在以下要求：name 和 birthDate 必须返回原始的不加修饰的值。也就是说，返回值不允许存在边界符号。或者说，如果一个人名字叫做荷马，那么调用这个人的名字函数应返回"荷马"而不应返回"[荷马]"。

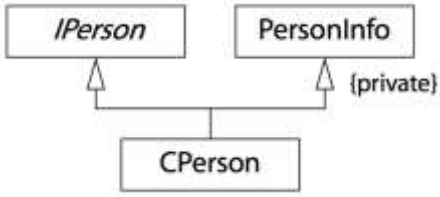
CPerson 和 PersonInfo 之间的关系应该是：PersonInfo 应该恰巧拥有一些函数使得 CPerson 更易于实现。仅仅如此。二者的关系即“A 以 B 的形式实现”，我们知道这一关系可以用两种方式来表达：通过组合（参见条目 38）和通过私有继承（参见条目 39）.条目 39 中指出一般情况下组合是更好的解决方案，但是如果需要重定义虚函数时，使用继承就是必须的。在这种情况下，由于 CPerson 需要重定义 valueDelimOpen 和 valueDelimClose，因此简单的组合方案无法满足要求。最符合直觉的方案就是让 CPerson 私有继承自 PersonInfo，尽管在条目 39 中解释过，如果多做出一点点努力，CPerson 也是能够通过配合使用组合和继承来高效的重定义 PersonInfo 的虚函数的，这里我们依然使用虚拟继承。

然而，CPerson 必须同样实现 IPerson 的接口，这就需要用到公共继承。多重继承的一个合理应用方案应运而生 :配合使用一个接口的公共继承和一个具体实现的私有继承：

```
class IPerson {                              // 该类确定了需要实现的接口
public:
  virtual ~IPerson();
  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};

class DatabaseID { ... };                    // 下文中使用，忽略细节

class PersonInfo {                           // 该类中包含实现 IPerson 接口
public:                                      // 需要用到的函数
  explicit PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  ...
private:
  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;
  ...
```

```
};

class CPerson: public IPerson, private PersonInfo {
public:                                     // 该类使用了 MI
  explicit CPerson(DatabaseID pid): PersonInfo(pid) {}

  virtual std::string name() const          // IPerson 成员函数必要的具体实现
  { return PersonInfo::theName(); }

  virtual std::string birthDate() const     // 同上
  { return PersonInfo::theBirthDate(); }
private:                                    // 继承的虚拟的分隔符号函数的重定义
  const char * valueDelimOpen() const { return ""; }
  const char * valueDelimClose() const { return ""; }
};
```

上述设计方案可描绘为以下 UML 图：



这个示例向我们展示了，MI 既可以有使用价值，也可以是完善的解决方案。

说一千道一万，多重继承仅仅是面向对象编程工具箱里的另一件工具罢了。与单继承相比，它一般都会更难使用且更难理解，因此如果你同时拥有一套 MI 和一套 SI 设计方案，而两者功能又基本等价时，SI 方案无疑是更好的选择。如果你能想出的唯一的解决方案中包含 MI，你应该再多加思索一下，几乎可以肯定存在某种方法来使用 SI。与此同时，某些时候 MI 是最清晰、最易于维护、最合理的解决方案。如果真是这样，那么不要感到害怕，只要审慎应用之即可。

时刻牢记

- 多重继承比单继承要复杂得多。它将导致新的歧义问题，并且会引入虚拟继承。

- 虚拟继承会带来空间、速度以及初始化和赋值的复杂度等多方面的额外开销。当虚拟基类为空时才最有实用价值。
- 多重继承确实有合理的应用场景，其中之一就是：当需要配合使用“对一个接口类的公共继承”和“对一个包含实现的辅助类的私有继承”时。

第七章. 模板和泛型编程

C++引入模板最初的动机简单明了，即让“创建诸如 vector、list 和 map 这样的类型安全的容器”成为可能。然而人们发现模板的应用范围远不止此。容器的确美妙，但泛型编程可以让代码无需与当前操作的对象类型相关，更是精美绝伦。诸如 for_each、find 和 merge 这样的 STL 算法就是这一编程手法的实例。最终，人们发现 C++的模板编程机制是图灵完备的：它可以用于计算所有可计算的值。由此可引入模板元编程的概念，运用这一概念的程序在 C++编译器内部执行，随编译结束而结束。 如果说 C++模板编程是弱水三千，容器则只当一瓢。虽然模板编程博大精深，仍存在一系列核心理念，即模板编程的基础。这些理念就是本章讨论的焦点。

本章不会试图让你成为一个模板编程专家，但是会让你更好的使用模板。同时还会为你提供一些必要的信息，从而扩展你对模板编程的认识边界，直至你想象力的边界。

条目41： 理解隐式接口和编译时多态

面向对象编程的世界基于**显式**接口和**运行时**多态运行。如下面代码中的类和函数（二者都没有具体含义）所示：

```
class Widget {
public:
  Widget();
  virtual ~Widget();
  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);        // 参见条目 25
  ...
};

void doProcessing(Widget& w) {
  if (w.size() > 10 && w != someNastyWidget) {
Widget temp(w);
```

```
temp.normalize();
temp.swap(w);
  }
}
```

对于 doProcessing 中的 w 我们可以这样描述：

- 由于 w 声明为 Widget 类型，因此 w 必须支持 Widget 的接口。我们可以在源代码（比如 Widget 的头文件）中找到这一接口的确切情况。因此我将其命名为显式接口——即在源代码中显式可见的接口。
- 由于 Widget 中的一些成员函数是虚函数，因此 w 对它们的调用将呈现出运行时多态：在运行时调用哪个特定的函数由 w 的动态类型决定。(参见条目 37)

模板与泛型编程的世界观与传统有本质的不同。在新世界里，显式接口和运行时多态依旧存在，但是它们的重要性减弱了。取而代之的是，**隐式接口**和**编译时多态**将大行其道。为了了解这一现象的成因，请观察，当我们把 doProcessing 从函数改写为函数模板时，发生了什么：

```
template<typename T> void doProcessing(T& w) {
  if (w.size() > 10 && w != someNastyWidget) {
    T temp(w);
    temp.normalize();
    temp.swap(w);
  }
}
```

现在，我们对于 doProcessing 中的 w 应该怎样描述呢？

- w 必须支持的接口是由模板内的 w 所进行的操作决定的。本示例中，可以看出 w 的类型（T）必须支持 size、normalize 和 swap 这些成员函数；拷贝构造函数（用来创建 temp）；和不等于的比较函数（用于同 someNastyWidget 进行比较）。稍后我们将介绍，这样做并不是十分精确，但是目前已足以符合要求了。重要的是：必须存在一组合法的表达式以使模板顺利通过编译，这组表达式就是 T 类型必须支持的隐式接口。

- 为确保涉及 w 的函数（比如 operator>和 operator!=）能够调用成功，可能会引入模板实例化。此类实例化操作发生在编译过程中。这是因为通过不同的模板参数来实例化函数模板将导致不同的函数被调用，这也就是所谓的“编译时多态”。

即使你从未使用过模板编程，你也应该对运行时和编译时多态不会陌生，这是因为二者之间的区别与“确认一组重载函数中应该调用哪一个（编译时进行）”和“虚函数调用的动态绑定（运行时进行）”之间的区别相仿。只是，显式接口和隐式接口对于模板是新的概念，需要进一步探究。

显式接口一般情况下由函数签名组成，也就是说，函数名字、参数类型和返回类型等等。比如 Widget 类的公共接口：

```
class Widget {
public:
  Widget();
  virtual ~Widget();

  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);
};
```

是由一个构造函数、一个析构函数，以及 size、normalize 和 swap 函数组成，还包括函数的参数类型、返回类型、它们是否是 const。（同时还包括由编译器自动生成的拷贝构造函数和拷贝赋值运算符——参见条目 5）还有可能包括 typedef，另外如果你冒失的将数据成员声明为公共的（这将违背条目 22 的建议），那么数据成员也包括在显式接口内，本示例中并没有这样做。

隐式接口则大不相同。它不基于函数签名存在，而是由逻辑**表达式**组成。请再次观察 doProcessing 模板开头的条件判断：

```
template<typename T> void doProcessing(T& w) {
  if (w.size() > 10 && w != someNastyWidget) {
    ...
```

T（w 的类型）的隐式接口看似受以下条件约束：

- 它必须提供一个返回整数值的成员函数 size。
- 它必须支持一个 operator!=函数来比较两个 T 类型的对象。（这里，我们假设 someNastyWidget 是 T 类型的。）

多亏了运算符重载，二者都不需要强制遵守。是的，T 必须提供一个 size 成员函数，尽管值得注意的是该函数可能继承自某个基类。但是该成员函数并非必须返回一个整数。返回值甚至不必是数值。进而，返回值类型甚至不必为 operator>的参数类型！只需返回一个 X 类型的对象，并且确保存在一个 opetator>可以被一个 X 对象和一个 int（因为 10 是一个 int）调用就行了。operator>也并非必须使用 X 类型的参数，只要 X 类型的对象可以隐式转换为 Y 类型的对象，operator>也可以使用 Y 作为参数类型。

类似地，T 也不是必须支持 operator!=，只要 T 可转换为 X 并且 someNastyWidget 可转换为 Y，operator!=就可以成功接受一个 X 类型的对象和一个 Y 类型的对象作为参数。

（旁白：上述的分析并没有考虑到 operator&&被重载的可能性，如果这一重载真的发生了，那么上述表达式将从与条件判断式做出潜在的巨大转变。）

大多数人在开始以这种方式思考隐式接口时都会感到头疼，但是你真的不需要吃阿司匹林。很简单，隐式接口就是由一组逻辑表达式组成。表达式本身可能看上去很复杂，但是它们所确保的约束条件一般都简单明了。比如，上文中的条件表达式：

```
if (w.size() > 10 && w != someNastyWidget) ...
```

很难确定 size、operator>、operato&&以及 operator!=所约束的内容，但是整个表达式的约束条件则十分明确。由于一个 if 语句的条件部分必须是一个布尔表达式，因此无论涉及哪些具体的类型，无论“w.size() > 10 && w != someNastyWidget”生成了什么，都必须与 bool 兼容。它是由 doProcessing 的类型参数 T 所确定的隐式接口的一部分。doPrecessing 所需的剩余部分接口即对拷贝构造函数、normalize 和 swap 函数的调用，这些调用对于 T 类型的对象是必须合法的。

“由模板的参数所确定的隐式接口”和“由类对象所确定的显式接口”同样是真实存在的，二者都在编译时得到检查。如果某个类的对象与类所提供的显式接口相矛盾，

那么你将无法使用这个对象（这样的代码不能通过编译），同理，如果一个由模板生成的对象不支持模板所需的隐式接口，那么你也无法使用这个对象（同样也无法通过编译）。

时刻牢记

- 类和模板都提供接口和多态特性。
- 类的接口是显式的，基于函数签名。多态通过虚函数产生于运行时。
- 模板参数的接口是隐式的，基于逻辑表达式。多态通过模板实例化和函数重载解析产生于编译时。

条目42： 理解 typename 的双重含义

问：下面的模板声明中，class 与 typename 有什么区别？

```
template<class T> class Widget;        // 使用"class"
template<typename T> class Widget;     // 使用"typename"
```

答：没区别。当你声明一个模板类型参数时，class 和 typename 的含义完全一致。一些程序员喜欢一直使用 class，因为在键盘上更好打一些。而另一些（包括作者）喜欢用 typename，因为它能够表明参数并非必须是 class。一些开发者在所有类型都允许使用时使用 typename，在仅允许使用用户自定义类型时才使用 class。但是从 C++的角度看，在声明模板参数时，class 和 typename 的含义完全相同。

然而 C++并非总将 class 和 typename 无差别对待。一些时刻你必须使用 typename。为了描述这"一些时刻"，我们必须先讨论一下在模板内可以引用的两种名字。

假设我们有一个函数模板，它取一个兼容 STL 的容器作为参数，容器中可存放可以赋值为 int 的对象。另外假设这个函数只是简单的打印出容器中的第二个元素。这是一个以"愚蠢"手法来实现的"愚蠢"函数，愚蠢到无法通过编译，但是请先忽略细节，以下是"愚蠢"的实现方法：

```
template<typename C>                              // 打印容器中第二个元素
void print2nd(const C& container)                 // 无效的C++代码！
{
  if (container.size() >= 2) {
C::const_iterator iter(container.begin());  // 取得首个元素的迭代器
++iter;                                           // 将iter移至第二个元素
    int value = *iter;                            // 将元素值复制到一个int中
std::cout << value;                               // 打印这个int
  }
}
```

我将此函数中的两个局部变量（iter 和 value）进行了加粗显示。iter 的类型是 C::const_iterator，这一类型是由模板参数 C 决定的。模板中从属于模板参数的名字称为“**从属名字**”。当一个从属名字嵌套于某个类中时，我将其称为“**嵌套从属名字**”。C::const_iterator 就是一个嵌套从属名字。实际上，它是一个“嵌套从属类型名字”，也就是一个为某个类命名的嵌套从属名字。

print2nd 另一个局部变量 value 是 int 类型的。int 是一个不从属于任何模板参数的名字。这样的名字被称为“非从属名字”。（我不知道为什么不把它们称为“独立名字”。如果你觉得很不适应“非从属名字”这个术语，那么可以说我和你所见略同了，但是“非从属名字”的确就是这一类名字的术语，因此你必须适应它们。）

嵌套从属名字可以导致语法分析的困难。比如说，假设我们通过以下的方法是的 print2nd 函数变得更加愚蠢：

```
template<typename C>
void print2nd(const C& container)
{
  C::const_iterator * x;
  ...
}
```

看上去我们将 x 声明为一个指向 C::const_iterator 局部指针变量。但是只有在我们事先知道 C::const_iterator 是一个类型时才成立。然而如果 C::const_iterator 不是一个类型呢？如果 C 正好拥有一个名为 const_iterator 的静态数据成员呢？如果 x 正好是一个全局变量的名字呢？在

这样的情况下，上述的代码将不会声明一个局部变量，而是对C::const_iterator和x进行一次乘法运算！的确，这听上去像是疯了，但是这确实是可能的，同时C++语法分析程序的编写者必须考虑到所有可能的输入情形，即使是疯了的情形。

如果我们不知道C的细节，那么我们就无法得知C::const_iterator到底是不是一个类型，当对print2nd模板进行语义分析时，C尚未浮出水面。C++按以下规则处理这个歧义：如果语义分析器在某个模板中遇到了一个嵌套从属名字，那么就假定这个名字不是类型，除非你明确指出。在默认情况下，嵌套从属名字不是类型名字。（此规则存在一处例外，稍后讲解。）

头脑中有了这些信息，请再次观察print2nd的起始部分：

```
template<typename C>
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    C::const_iterator iter(container.begin());
    ...                              // C++假定这个名字不是一个类型名
  }
}
```

现在这段代码是无效C++代码的原因就非常明显了。iter的声明只有在C::const_iterator是一个类型的情况下有意义。但是我们并没有明确告诉C++这一点，此时C++就假定它不是类型。为了修正这一状况，我们必须告知C++，C::const_iterator是一个类型名。通过在它前面添加typename来实现：

```
template<typename C>                 // 合法的C++代码
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());
    ...
  }
}
```

这里的一般规则十分简单：每当你在模板内使用一个嵌套从属类型名字时，你必须在这个名字前面添加typename关键字。(再次强调，稍后我将讲解一个例外情形。)

typename应该仅用于辨别嵌套从属类型名字，其他名字均不适用。比如，下面是一个函数模板，它取一个模板和一个针对该模板的迭代器作为参数：

```
template<typename C>                    // 允许使用typename（和class等价）
void f(const C& container,              // 禁止使用typename
       typename C::iterator iter);      // 必须使用typename
```

由于C不是一个嵌套从属类型名字（它没有内嵌于任何从属于模板参数的东西），因此在声明container时必须不能添加typename关键字，但是由于C::iterator是一个嵌套从属类型名字，因此应该在它前面添加typename关键字。

“必须在嵌套从属类型名之前使用typename”规则存在的特殊情形即：在基类列表中，或在一个成员初始化列表中作为一个基类标识时，绝不要在嵌套从属类型名之前使用typename。举例说明：

```
template<typename T>
classDerived:public Base<T>::Nested { // 基类列表：禁止使用typename
public:
  explicit Derived(int x)
  : Base<T>::Nested(x)                // 成员初始化表中的基类标识：
  {                                   // 禁止使用typename

    typename Base<T>::Nested temp;    // 除了基类表和成员初始化表中的基类
    ...                               // 标识之外的嵌套从属类型名字，必须
  }                                   // 使用typename
  ...
};
```

C++如此飘忽不定的特性令人郁闷。但是一旦拥有了一定的经验之后，你几乎都不会注意到这一点。

上文的typename示例对于你在实战中会遇到的某些情形有一定代表意义，因此让我们再次观察一下。假定我们正在编写一个函数模板，它以一个迭代器作为参数，

同时我们期望生成一个此迭代器所指向的对象的局部副本，名为 temp。我们可以这样做：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typename std::iterator_traits<IterT>::value_type temp(*iter);
  ...
}
```

不要被代码中的 std::iterator_traits<IterT>::value_type 给吓到。这只是一次标准库中 traits 类的一般应用（参见条目 47）。C++描述“IterT 类型的对象所指向的东西的类型”的方法。这一语句声明了一个与 iter 所指的对象类型相同的局部变量（temp），并且它使用 iter 所指的对象来初始化 temp。如果 IterT 是 vector<int>::iterator，那么 temp 就是 int 类型。如果 IterT 是 list<string>::iterator，那么 temp 就是 string 类型。

由于 std::iterator_traits<IterT>::value_type 是一个嵌套从属类型名字（value_type 是内嵌于 iterator_traits<IterT>中的，并且 IterT 是一个模板参数），因此我们必须对其使用 typename。

如果你对于阅读 std::iterator_traits<IterT>::value_type 感到不适，那么录入时的感觉便可想而知。如果你像大多数程序员一样，你会感觉多次录入它们是可怕的，此时你需要创建一个 typedef。对于类似于 value_type 这样的 traits 成员名称（再次提醒，对于 traits 成员和 traits 类的信息请参见条目 47），可以使用 traits 成员的名字作为 typedef 的名字（这是一个惯例），因此这样的局部 typedef 一般这样定义：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typedef typename std::iterator_traits<IterT>::value_type value_type;
  value_type temp(*iter);
  ...
}
```

大多数程序员都会发现 typedef 与 typename 并列出现显得很不协调，但是这

是遵守调用**嵌套从属类型名字**的规则而产生的（合理的）后果。你会很快的适应它的。毕竟这里你有着强烈的动机。你也不想录入那么多遍的 `typename std::iterator_traits<IterT>::value_type`，不是吗。

作为补全，还需强调一点：不同的编译器遵守 `typename` 相关规则的程度是不同的。一些编译器对于必须使用 `typename` 但是没有使用的代码也能够接受；而一些编译器对于禁止使用 `typename` 但是却使用了的代码也没有问题；而一少部分编译器（通常是一些老版本的）则会在已经存在并且必须使用 `typename` 的情况下报错。这也就意味着 `typename` 和嵌套从属类型名字之间的关系将会为代码的可移植性带来一点点的麻烦。

时刻牢记

- 在声明模板参数时，`class` 和 `typename` 可以互换。
- 使用 `typename` 来指定嵌套从属类型名字。除了在基类列表和在成员初始化表中充当基类标识的情形。

条目43： 了解如何访问模板化基类内的名字

假设我们需要编写一个可以为几个不同的公司发送消息的应用程序。这些消息的格式可以是加密的，也可以是明文（非加密的）。如果在编译时我们对于“哪条消息要发送给哪个公司”有足够的信息，那么我们可以使用以下的基于模板的解决方案：

```
class CompanyA {
public:
  ...
  void sendCleartext(const std::string& msg); // 发送明文
  void sendEncrypted(const std::string& msg); // 发送密文
  ...
};

class CompanyB {
public:
  ...
```

```
  void sendCleartext(const std::string& msg);  // 同上
  void sendEncrypted(const std::string& msg);  // 同上
  ...
};
...                                            // 其他公司的类

class MsgInfo { ... };                         // 保存信息的类
                                               // 用于创建消息
template<typename Company>
class MsgSender {
public:
  ...                                          // 构造函数、析构函数等
  void sendClear(const MsgInfo& info)
  {
std::string msg;
使用信息 info 创建消息 msg;

    Company c;
c.sendCleartext(msg);
  }

  void sendSecret(const MsgInfo& info)         // 与 sendClear 类似，只是这
  { ... }                                      // 里调用 c.sendEncrypted
};
```

上述代码可以顺利运行，但是假设我们在某些情况下需要在每次发送消息时向日志中添加一些信息。很简单，可以通过派生类来添加这一功能，而且这看上去也是一个可行的解决方案：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    将“发送前”的信息添加进日志;
```

```
    sendClear(info);                    // 调用基类函数（这段代码不能通过编译！）
    将"发送后"的信息添加进日志;
  }
  ...
};
```

请注意派生类中发送消息的函数(sendClearMsg)与基类(函数名为sendClear)拥有不同的名字。这是一个优秀的设计，因为这样做避开了派生类中的名字隐藏问题（参见条目 33），以及"重定义派生的非虚函数"这一内在问题（参见条目 36）.但是上述代码是不会通过编译的，至少遵循标准的编译器不会。这样的编译器将会报错"sendClear 不存在"。我们可以看到 sendClear 存在于基类里，但是编译器却不会去那里查找它。我们需要理解这一现象的原因。

问题在于：当编译器遇到类模板 LoggingMsgSender 时，它们无从得知这个类模板的基类到底是什么。的确，这里是 MsgSender<Company>，但是 Company 是一个模板参数，不到某个时刻（当 LoggingMsgSender 被实例化时）编译器就无从了解它的含义。在不知道 Company 是何方神圣的情况下，MsgSender<Company>类的具体含义就无从谈起了。特别地，我们也就无法知道 sendClear 是怎样的函数。

为了明确的说明这一问题，我们假设存在一个 CompanyZ 类，它只能进行加密通信：

```
class CompanyZ {                        // 这个类不提供 sendClearText 函数
public:
  ...
  void sendEncrypted(const std::string& msg);
  ...
};
```

泛化的 MsgSender 对于 CompanyZ 就不适用了，这是因为该模板提供的 sendClear 函数对于 CompanyZ 类没有任何意义。为了纠正这一问题，我们可以为 CompanyZ 创建一个特化版本的 MsgSender 模板：

```
template<>                              // MsgSender 的一个全特化版本与一般的模板
class MsgSender<CompanyZ> {             // 相同，但剔除了 sendClear
```

```
  public:
  ...
void sendSecret(const MsgInfo& info)
  {...}
};
```

请注意类定义开头的“template<>”语法。它意味着所定义的东西既不是模板也不是一个独立的类。而是模板 MsgSender 在模板参数为 CompanyZ 时的一个特化版本。即所谓的“**模板全特化**”：MsgSender 为 CompanyZ 类型特化了，并且特化是完全的，一旦类参数定义为 CompanyZ，模板参数就再也不能更改了。

由于 MsgSender 已经为 CompanyZ 进行了特化，请再次考虑派生类 LoggingMsgSender：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
  public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    将“发送前”的信息添加进日志;
    sendClear(info);                    // 如果 Company == CompanyZ
                                        // 这个函数就不存在！
    将“发送后”的信息添加进日志;
  }
  ...
};
```

就像注释中所描述的，上述代码在基类是 MsgSender<CompanyZ>时将不成立，这是因为这个类中没有提供 sendClear 函数。这也就解释了为什么 C++不接受这样的调用：C++意识到基类模板可能会被特化，并且这样的特化版本可能会与泛化的模板拥有不同的接口。因此在一般情况下，C++将拒绝为模板化的基类去查找派生的名字。从某种意义上讲，当我们从面向对象的 C++转向模板 C++时（参见条目 1），继承的守则会失效。

为了重新开展这一工作，我们必须用某种手段来禁用 C++的“不要查找模板化基类”

行为。这里有三种方案可供选择。第一，你可以在对基类函数的调用前面添加“this->”：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
  public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    将“发送前”的信息添加进日志;
    this->sendClear(info);          // 工作正常。
                                    // 假定 sendClear 将被继承
    将“发送后”的信息添加进日志;
  }
  ...
};
```

第二，你可以使用一个 using 声明，如果你读过条目 33 的话你将对此解决方案感到熟悉。条目 33 中解释了 using 声明如何能够将隐藏的基类名字带入派生类的作用域中。因此我们可以这样编写 sendClearMsg：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
  public:
  using MsgSender<Company>::sendClear;
  ...                               // 告知编译器请假定基类中存在 sendClear
  void sendClearMsg(const MsgInfo& info)
  {
    将“发送前”的信息添加进日志;
    sendClear(info);                // 工作正常。
                                    // 假定 sendClear 将被继承
    将“发送后”的信息添加进日志;
  }
  ...
};
```

（尽管在这里以及条目 33 中，使用 using 声明都可以达到各自的目的，但是二者

所面临的问题是不同的。这里所面对的情况并不是“基类的名字被派生类名字所隐藏”，而是“除非我们告知编译器去查找基类作用域，否则它不会这样做。”）

让你的代码得以通过编译的第三个方法，是显式指明当前正在调用的函数是基类中的：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    MsgSender<Company>::sendClear(info);   // 工作正常
    ...                                    // 假定 sendClear 会被继承
  }
  ...
};
```

一般情况下，不到万不得已不要使用第三种方法，因为一旦调用的是虚函数，那么显式限定将使虚拟绑定行为失效。

从名字可见度的角度出发，上述每个解决方案都做了同一件事，即：向编译器做出保证，基类模板所有衍生的特化版本都能够支持由泛化的模板提供的接口。在进行类似 LoggingMsgSender 的派生类模板的语义分析时，所有的编译器都需要得到这样的保证。但是如果该保证确未实现，真相将在特化编译过程中浮出水面。举例说，如果后续的代码包含以下内容：

```
LoggingMsgSender<CompanyZ> zMsgSender;
MsgInfo msgData;
...                                  // 将信息放置在 msgData 内，
zMsgSender.sendClearMsg(msgData);    // 错误！无法通过编译
```

对于 sendClearMsg 的调用将无法通过编译，这是因为此时编译器知道基类是一个模板特化版本 MsgSender<CompanyZ>，同时编译器还知道这个基类不会提供 sendClearMsg 尝试调用的 sendClear 函数。

基础层面，这里的问题就是“编译器将在什么时机诊断对基类成员是否存在非法引用呢，是更早些时（在对派生类模板定义进行语义分析时）还是更晚些时（在模板已经使用特定的模板参数进行实例化后）呢”。C++的策略是使用更早期的诊断，这也就是为什么对于由模板实例化的类，C++会假设对它们基类的内容完全不知晓。

时刻牢记

- 要在派生类模板中调用基类模板内部的名字，可以通过“`this->`”前缀，通过`using`声明，或者通过一个显式的基类限定来实现。

条目44： 将参数无关的代码从模板中分离

模板是节约时间和避免代码重复的一个美妙的手法。你不再需要“录入 20 个类似的类，每个类有 15 个成员函数”，你只需要录入一个类模板，然后让编译器来实例化出 20 个特化的类和所需的 300 个函数。（由于类模板的成员函数仅当被使用时才会被隐式实例化，因此只有所有函数都被实际调用时你才会得到所有的 300 个成员函数。）函数模板同样神奇。你不需要编写许多函数，只需要编写一个函数模板然后把剩下的工作交给编译器就行了。科技是不是很炫酷？

的确，呃……有些时候是的。如果你不够留心，使用模板将导致**代码膨胀**（即完全重复（或几乎完全重复）的源代码或数据所组成的目标代码）。结果可能是这样：源代码看上去瘦而美，而目标代码却痴而肥，痴肥很难受欢迎的，因此你需要了解如何避免此类的目标代码泡沫。

主要的工具拥有一个高大上的名字——“**共性和特性分析**”，但是工具的理念并没有那么高大上。即使你过去从未编写过模板，你随时都在进行着这一分析。

当你正在编写一个函数，并且你发现这个函数实现的某些部分基本上与另一个函数中的相同，那是不是该复制粘贴？当然不是。你将两个函数共性的代码分离出来，并将它们放置在第三个函数里，然后让原先的两个函数来调用这个新的函数。这就是说，你分析了两个函数得到它们之间的共性部分和特性部分，然后你将共性部分放置在一个新函数中，让特性部分保留在原始的函数中。类似的，如果你正在编写一个类，你意识到类中某部分与其他类中的相同，那么你便无需复制共性部分。只

需将共性部分移入一个新类中，然后使用继承或组合（参见条目 32、38 和 39）让原始类访问共性内容。原始类中不同的部分（也就是特性的部分）留在远处。

编写模板时为了避免重复可如法炮制，但此处存在一个例外。在非模板代码中，重复是显式的，你可以**看到**两个函数或两个类之间的重复。在模板代码中，重复是隐式的：模板源代码只有一份，因此你需要训练自己在模板多次实例化时感知重复的出现。

举例说，你希望编写一个模板，用来存放固定大小的方形矩阵，并支持转置。

```
template<typename T, std::size_t n>    // 用来存放 n 乘 n 个 T 对象的模板
class SquareMatrix {                   // 关于参数 size_t，参见下文
public:
  ...
  void invert();                       // 就地转置
};
```

此模板有一个类型参数 T，但是它还有一个**非类型参数** `size_t`。非类型参数比起类型参数更少见，但是它们完全合法，在同时本示例中还非常自然。

请考虑以下代码：

```
SquareMatrix<double, 5> sm1;
...
sm1.invert();                      // 调用 SquareMatrix<double, 5>::invert
SquareMatrix<double, 10> sm2;
...
sm2.invert();                      // 调用 SquareMatrix<double, 10>::invert
```

`invert` 将被实例化两次。且函数并不完全相同，因为其中一个基于 5×5 的矩阵而另一个基于 10×10 的矩阵，不考虑常数 5 和 10，两个函数是一致的。这是模板导致的代码膨胀的一种典型方式。

两个函数除了一个 5 和一个 10 以外每个字母都是一样的，看到它们你作何感想？你的直觉告诉你应该创建一个函数，并让这个数值作为一个参数，然后使用 5 或 10 去调用这个参数化的函数，而不是直接复制代码。你的直觉是正确的。以下是 `SquareMatrix` 的修订版：

```
template<typename T>                        // 方形矩阵的基类，大小无关
class SquareMatrixBase {
protected:
  ...
  void invert(std::size_t matrixSize);  // 转换尺寸为 size 的矩阵
  ...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::invert;     // 让 invert 的基类版本在此派生类中可见
                                         // 参见条目 33 和 43
public:
  ...
  void invert() { invert(n); }           // 内联调用 invert 的基类版本
};
```

如你所见，invert的参数化版本存放于SquareMatrixBase基类中。和SquareMatrix一样，SquareMatrixBase也是一个模板，但是与SquareMatrix不同的是，它只将矩阵中对象的类型进行了模板化，不涉及矩阵的大小。因此，所有给定对象类型的矩阵将共用一个SquareMatrixBase类。于是它们将共用同一个基类版本的invert。(当然，请不要将此函数内联。如果这样做，SquareMatrix::invert的每次实例化都将得到一个SquareMatrixBase::invert代码的副本（参见条目30），届时你又将重陷代码重复的泥沼。)

在派生类中防止代码重复似乎是SquareMatrixBase::invert存在的唯一理由，因此它应是protected而不是public。由于派生类的invert内联调用基类版本，因此调用不会带来任何额外开销。(隐式inline，参见条目30。）另请注意SquareMatrix和SquareMatrixBase之间使用了私有继承。这精确地反映了基类只是派生类实现的促成手段，并非表达SquareMatrix与SquareMatrixBase之间在概念上存在“A是一个B”的关系。（私有继承的信息请参见条目39。）

一切按部就班，但此处还有一个棘手的问题我们尚未提及。SquareMatrixBase::invert 如何知道它所操作的数据类型呢？它从参数得知了矩阵的大小，但是它从哪里获取具体矩阵数据的类型呢？可能只有派生类知道了。而派生类又如何与基类通信以便基类完成转置操作呢？

为 SquareMatrixBase::invert 添加另一个参数似乎是一个可行的办法，参数可以是一个指向矩阵数据所在的内存块首地址的指针。这或许可行，但很可能 invert 并不是 SquareMatrix 内唯一大小不相关的、可转移至 SquareMatrixBase 中的函数。如果这样的函数存在若干，那么它们就都需要获取保存矩阵数据的内存地址。我们可以为每一个函数再添加一个参数，但是我们就需要为 SquareMatrixBase 反复提供同样的信息。这看上去不妙。

另一个选择是让 SquareMatrixBase 保存一个指针来指向矩阵数据的内存。只要指针不丢，它就可以同时保存矩阵的大小。由此可引出以下设计方案：

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T *pMem)  // 保存矩阵大小和一个
  : size(n), pData(pMem) {}                 // 指向矩阵数据的指针

  void setDataPtr(T *ptr) { pData = ptr; }  // 重设 pData
  ...
  private:
    std::size_t size;                       // 矩阵的大小
    T *pData;                               // 指向矩阵数据的指针
};
```

这样派生类需要自行决定如何分配内存。一些实现可能会选择将矩阵数据直接保存在 SquareMatrix 对象内部：

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()
  : SquareMatrixBase<T>(n, data) {}    // 将矩阵大小和数据指针传回基类
```

```
  ...
private:
  T data[n*n];
};
```

尽管此类对象不需要动态内存分配，但是对象的数量可能非常庞大。似乎可以转而将每个矩阵的数据放置在堆上：

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()                          // 将基类指针设为 null，
  : SquareMatrixBase<T>(n,0),             // 为矩阵数据分配内存，
    pData(new T[n*n])                     // 保存一个指针指向该段内存
  { this->setDataPtr(pData.get()); }      // 将指针的副本返回基类
  ...
private:
  boost::scoped_array<T> pData;           // 关于 boost::scoped_array
};                                        // 请参见条目 13
```

无论数据保存在何处，关键的结果是，从代码膨胀的角度看，现在许多（可能是所有）SquareMatrix 的成员函数可以通过简单的内联调用访问（非内联的）基类版本，而基类版本的函数对所有矩阵数据类型都适用，无论矩阵大小。与此同时，不同尺寸的 SquareMatrix 对象之间类型不同，因此即使 SquareMatrix<double, 5>和 SquareMatrix<double, 10>的对象均使用了 SquareMatrixBase<double> 中同样的成员函数，也无法将一个 SquareMatrix<double, 5>对象传递给以 SquareMatrix<double, 10>为参数的函数。是不是很不错？

的确很不错，但并不完美。内嵌了矩阵尺寸的 invert 函数的版本应该可以生成比将尺寸作为一个函数参数或直接保存在对象中的版本更好的代码。比如，在给定尺寸的版本中，版本信息将作为一个编译时常量，于是可以做出编译优化，比如常量传播，包括将其折叠在类似直接操作数的生成指令中。这一点尺寸不相关的版本无法做到。

另一个需要考虑的效率因素涉及对象的尺寸。如果你不够小心，将尺寸不相关的函

数移入基类，便可能增大每个对象的总体尺寸。比如，在上文的代码中，即使每个派生类都有办法获得数据，每个 SquareMatrix 对象仍有一个指向 SquareMatrixBase 类数据的指针。这就使每个 SquareMatrix 对象至少增大了一个指针的尺寸。修改设计以排除这些指针是可能的，但是再次强调，这里需要权衡考虑。比如，让基类保存一个指向矩阵数据的 protected 指针会削弱封装度（参见条目 22）。同时也会带来资源管理的难题：如果基类保存了一个指向矩阵数据的指针，但是这些数据既非动态分配，也未在派生类对象中物理保存（就像前文看到的），那么如何确定这个指针是否应该得到释放呢？这类问题是有答案的，但是你考虑越深，事情就变得越复杂。此时，一点小小的代码重复也是一种解脱了。

本条目只讨论了由非类型模板参数产生的代码膨胀，但是类型参数也会导致膨胀。比如，在许多平台，int 和 long 在目标代码中的表示形式一致，因此 vector<int> 和 vector<long>的成员函数很可能是一致的，这恰恰就是代码膨胀的定义。一些连接器会将一致的函数实现合并，但是一些不会，这就意味着一些同时实例化出 int 和 long 版本的模板可能在某些环境中导致代码膨胀。类似地，在大多数平台中，所有指针类型在目标代码中表示形式均一致，因此包含指针类型（比如 list<int*>、list<const int*>、list<SquareMatrix<long, 3>*>，等等）的模板应该经常能够为每个成员函数使用单一的基础实现。典型情况下，意味着通过调用一个使用无类型指针（也就是 void*指针）的函数来实现使用强类型指针（也就是 T*指针）的成员函数。C++标准库的一些实现中对于 vector、deque 和 list 等模板就是这样做的。如果你关心模板中所出现的代码膨胀问题，你也应该使用同样的方法设计模板。

时刻牢记

- 模板会生成多个类和多个函数，因此每段不依赖模板参数的模板代码都会导致代码膨胀。
- 由非类型的模板参数带来的膨胀可排除，通常用函数参数或类数据成员取代模板参数来进行。
- 由类型参数带来的膨胀可减少，可以通过为拥有一致目标代码表现形式的实例化类型共享实现来进行。

条目45： 使用成员函数模板来接纳“所有兼容类型”

智能指针是一类对象，它们的行为与指针基本一致，却提供了指针所不具备的功能。比如，条目 13 解释了标准库中的 auto_ptr 和 tr1::shared_ptr 可以在恰当的时候自动释放基于堆的资源。STL 容器的迭代器几乎总是智能指针；当然你不能期望通过“++”来移动链表中指向某节点的内建指针，但是对 list::iterator 可行。

支持隐式转换是实在指针所擅长的技能之一。派生类指针可隐式转换为基类指针，指向非 const 对象的指针可转换为指向 const 对象的指针，等等。请看以下示例中一个三层结构中发生的转换：

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;            // 转换：Middle* ⇒ Top*
Top *pt2 = new Bottom;            // 转换：Bottom* ⇒ Top*
const Top *pct2 = pt1;            // 转换：Top* ⇒ const Top*
```

在用户自定义的智能指针类中模拟类似转换是很困难的，我们需要保证下面的代码通过编译。

```
template<typename T>
class SmartPtr {
public:
  explicit SmartPtr(T *realPtr);   // 智能指针一般通过内建指针初始化
  ...
};

SmartPtr<Top> pt1 =               // 转换：SmartPtr<Middle> ⇒
  SmartPtr<Middle>(new Middle);   //       SmartPtr<Top>
SmartPtr<Top> pt2 =               // 转换：SmartPtr<Bottom> ⇒
  SmartPtr<Bottom>(new Bottom);   //       SmartPtr<Top>
SmartPtr<const Top> pct2 = pt1;   // 转换：SmartPtr<Top> ⇒
                                  //       SmartPtr<const Top>
```

由于同一个模板的不同实例之间不存在继承关系，因此编译器将SmartPtr<Middle>和SmartPtr<Top>视为完全不同的类，也就是说在编译器眼里它们不会比 vector<float>和 Widget 之间的关系更接近。为了确保SmartPtr 各类之间的转换正常进行，我们要手动编写程序。

在上文的智能指针示例中，每条语句都创建了一个智能指针对象，因此现在我们将致力于编写符合要求的智能指针构造函数。一个关键的结论是我们无法做到面面俱到。在上文的层次结构中，我们可以通过一个 SmartPtr<Middle>或一个SmartPtr<Bottom>来构造一个 SmartPtr<Top>，但是如果未来层次结构得到了扩展，其他智能指针类性也应能够构造 SmartPtr<Top>对象。例如，如果我们添加了：

```
class BelowBottom: public Bottom { ... };
```

程序应该支持“使用 SmartPtr<BelowBottom>对象来创建 SmartPtr<Top>”，但显然我们不希望通过修改 SmartPtr 来实现。

理论上，我们需要无数个构造函数。由于模板可以通过实例化生成无数个函数，因此我们似乎并不需要 SmartPtr 的构造**函数**，我们需要一个构造**模板**。此类模板叫做**成员函数模板**（亦称**成员模板**，可为一个类生成成员函数）。一个应用实例：

```
template<typename T>
class SmartPtr {
public:
  template<typename U>                    // 成员模板：“泛化拷贝构造函数”
  SmartPtr(const SmartPtr<U>& other);
  ...
};
```

这意味着对于每个 T 类型和每个 U 类型来说，每个 SmartPtr<T>都应能够由SmartPrt<U>来创建，因为 SmartPtr<T>有一个含 SmartPtr<U>参数的构造函数。这样的构造函数是通过同一模板实例化出的其它类型的对象来创建对象的（比如，通过 SmartPtr<U>来创建 SmartPtr<T>），可称为**泛化拷贝构造函数**。

上文的泛化拷贝构造函数并没有声明 explicit。这是有意为之的。在内建指针类型之间的类型转换（比如，从派生类到基类的指针）是隐式的，不需要转型，因此

有理由使用智能指针来模拟这一行为。为该模板化构造函数去掉 explicit 恰恰完成了这一工作。

泛化拷贝构造函数可以提供额外的功能。的确，我们需要通过 SmartPtr<Bottom>来创建 SmartPtr<Top>，但是我们不需要通过 SmartPtr<Top>来创建 SmartPtr<Bottom>，因为这是与公有继承的定义相违背的（参见条目 32）。我们同样不需要通过 SmartPtr<double>创建 SmartPtr<int>，因为并不存在从 double*到 int*的隐式转换方式。某种意义上讲，我们应对该成员模板生成的成员函数做出甄别，以做到优胜劣汰。

假设 SmartPtr 通过提供由智能指针对象保存的内建指针的副本来保证与 auto_ptr 和 tr1::shared_ptr 行为一致（参见条目 15），我们可以使用构造函数模板的实现将类型转换限制在所需的范围：

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other)     // 通过其他 heldPtr 来
  : heldPtr(other.get()) { ... }         // 初始化此 heldPtr

  T* get() const { return heldPtr; }
  ...

private:
  T *heldPtr;                            // 由 SmartPtr 保存的内建指针
};
```

我们在成员初始化列表中用 SmartPtr<U>保存的 U*类型指针来初始化 SmartPtr<T>的 T*类型数据成员。这只有在 U*指针可以隐式转换为 T*指针时成立，这正是我们所需要的。所带来的净效应即：SmartPtr<T>现在有一个泛化拷贝构造函数，只有在传入一个兼容类型的参数时才能通过编译。

成员函数模板的实用价值并不限于构造函数。它们的另一个常见作用就是对赋值的支持。比如，TR1 的 share_ptr（参见条目 13）支持通过所有兼容的内建指针、tr1::shared_ptr、auto_ptr 和 tr1::weak_ptr（参见条目 54），以及所

有被赋值为内建指针、tr1::shared_ptr 或 auto_ptr 的对象来进行构造。以下是 TR1 规范说明中对于 tr1::shared_ptr 的一段摘录，也可以看到标准在声明模板参数时更偏向使用 class 而不是 typename。（条目 42 中解释过此上下文环境中二者并无不同。）

```
template<class T> class shared_ptr {
public:
  template<class Y>                                  // 通过所有兼容内建指针、
    explicit shared_ptr(Y * p);                      // shared_ptr、
  template<class Y>                                  // weak_ptr 或
    shared_ptr(shared_ptr<Y> const& r);              // auto_ptr 的对象
  template<class Y>                                  // 进行构造
    explicit shared_ptr(weak_ptr<Y> const& r);
  template<class Y>
    explicit shared_ptr(auto_ptr<Y>& r);

  template<class Y>                                  // 从所有兼容 shared_ptr
    shared_ptr& operator=(shared_ptr<Y> const& r); // 或 auto_ptr 的对象
  template<class Y>                                  // 取得赋值
    shared_ptr& operator=(auto_ptr<Y>& r);
  ...
};
```

除泛化拷贝构造函数外的构造函数均为 explict。这意味着允许发生从 share_ptr 类型向其他类型的隐式转换，但从内建指针到其他智能指针类型的**隐式**转换是不允许的。（通过转型实现的**显式**转换是可以的。）另外有趣的是传递给 tr1::shared_ptr 的构造函数和赋值运算符的对象中，auto_ptr 没有声明为 const，而 tr1::shared_ptr 和 tr1::weak_ptr 却相反。这是因为“auto_ptr 在复制过程中做出修改时是独立的”（参见条目 13）。

成员函数模板是件美妙的东西，但是它们并未撼动语言的基本原则。条目 5 中解释了编译器可能自动生成四个函数，其中两个是拷贝构造函数和拷贝赋值运算符。tr1::shared_ptr 声明了一个泛化拷贝构造函数，很明显当 T 与 Y 类型相同时，泛化拷贝构造函数将会被实例化并创建“一般的”拷贝构造函数。因此在 tr1::shared_ptr 对象是由同类型的另一个 tr1::shared_ptr 对象构造得来

时，编译器是会生成一个 tr1::shared_ptr 版本的构造函数呢，还是会生成泛化拷贝构造函数的模板呢？

如我所说，成员模板并不会撼动语言的原则。原则指出如果在需要一个拷贝构造函数时你却没有声明它，那么编译器会自动为你生成一个。在一个类中声明一个泛化拷贝构造函数（成员模板）并不会阻碍编译器声称自己版本的拷贝构造函数（一个非模板的版本），因此如果你希望控制拷贝构造的方方面面，你必须同时自行声明一个泛化拷贝构造函数和一个“一般的”拷贝构造函数。同样的推理适用于赋值。以下是 tr1::shared_ptr 定义的一段节选，也解释了这一做法：

```
template<class T> class shared_ptr {
public:
  shared_ptr(shared_ptr const& r);                  // 拷贝构造函数

  template<class Y>                                 // 泛化拷贝构造函数
    shared_ptr(shared_ptr<Y> const& r);

  shared_ptr& operator=(shared_ptr const& r);       // 赋值运算符

  template<class Y>                                 // 泛化赋值运算符
    shared_ptr& operator=(shared_ptr<Y> const& r);
  ...
};
```

时刻牢记

- 使用成员函数模板来生成所需的函数以接收所有兼容类型。
- 如果你为泛化拷贝构造函数或泛化赋值运算符声明成员模板，也需声明一般化的拷贝构造函数和拷贝赋值运算符。

条目46： 在需要进行类型转换时要将非成员函数定义在模板内部

条目 24 解释了为什么当函数的所有参数需要隐式类型转换时只可使用非成员函数，同时条目中还使用了一个“有理数 Rational 类的 operator*函数”示例。我建议

你在继续阅读之前重温一下那个示例，因为本条目将通过对 Rational 和 operator*模板化对条目 24 的示例做出一个看似无害的修改，从而扩充该讨论主题。

```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0,     // 现在参数通过引用方式传递
           const T& denominator = 1); // 理由参见条目 20

  const T numerator() const;           // 返回值仍通过传值方式返回，参见条目 28
  const T denominator() const;         // 返回值为 const，参见条目 3
  ...
};

template<typename T>
const Rational<T> operator*(const Rational<T>& lhs,
                            const Rational<T>& rhs)
{...}
```

与条目 24 类似，由于我们希望支持混合模式的计算，因此我们希望下方的代码能够通过编译。我们期望能够成功，这是因为我们使用了与条目 24 中相同的代码。唯一的不同就是 Rational 和 operator*现在是模板：

```
Rational<int> oneHalf (1, 2) ;          // 条目 24 示例中 Rational 的模板版本
Rational<int> result = oneHalf * 2;     // 错误！不能通过编译
```

编译失败的事实表明了模板化的 Rational 与非模板化的版本存在着某些不同，不同确实存在。在条目 24 中，编译器知道我们试图调用什么函数（operator*取两个 Rational 参数），但是此处编译器对于需要调用的函数并不**知晓**。取而代之的是，它们试图通过名为 operator*的模板来**推断出**需要实例化（也就是创建）什么函数。它们知道即将实例化一个包含两个 Rational<T>参数的、名为 operator*的函数，但是为了完成这一实例化，它们还需要推断出 T 是什么。问题是它们做不到。

在尝试推断 T 时，编译器对于调用 operator*时传入参数的类型进行了检查。此

情形中的类型是 Rational<int>（oneHalf 的类型）和 int（2 的类型）。下面对每个参数单独讨论。

oneHalf 的推导很简单。operator*的第一个参数声明为 Rational<T>类型，而传入 operator*的第一个实际参数（oneHalf）类型为 Rational<int>，因此 T 必为 int。不幸的是，其他参数的推断就没有这么简单了。operator*的第二个参数声明为 Rational<T>类型，但是传入 operator*的第二个实际参数(2)是 int 类型的。在此情况下编译器如何推断出 T 是什么类型呢？你可能期望它们使用 Rational<int>的非 explict 构造函数将 2 转换为 Rational<int>，这样编译器就可以推断出 T 的类型是 int，但是它们并不会这样做，这是因为隐式类型转换函数**永远**不能胜任模板实在参数的推断过程。此类转换的确应用于函数调用的过程，但是在你可以调用一个函数之前，你必须事先了解存在哪些函数。为了做到这一点，你必须对相关函数模板的参数类型做出推断（以便实例化恰当的函数）。但是在模板实在参数推断中构造函数的隐式类型转换是无法胜任的。由于条目 24 不涉及模板，因此不存在模板实在参数推断问题。但是现在我们正在使用的是 C++的模板部分（参见条目 1），该问题则变成了首要问题。

模板类中的 friend 声明可以制定具体的函数，利用这一事实我们可为编译器解决“模板实在参数推断的问题”。这意味着 Rational<T>可将参数为 Rational<T>的 operator*声明为友元函数。类模板不依赖于模板实在参数推断（只有函数模板存在这一操作），因此在 Rational<T>被实例化时 T 的类型便是已知的。将 Rational<T>类中恰当的 operator*函数声明为友元可使问题简化：

```
template<typename T>
class Rational {
public:
  ...
  friend                              // 声明 operator*函数（细节见下文）
    const Rational operator*(const Rational& lhs,
                             const Rational& rhs);
};

template<typename T>                  // 定义 operator*函数族
const Rational<T>operator*(const Rational<T>&lhs,
```

```
                                const Rational<T>& rhs)
{...}
```

现在对 operator*的混合模式调用可以通过编译，这是因为当 oneHalf 对象声明为 Rational<T>类型时，Rational<int>将实例化，作为此过程的一部分，包含 Rational<int>参数的友元函数 operator*将被自动声明。作为一条**函数**（而不是函数**模板**）声明，编译器就可以在调用时使用隐式转换函数（比如 Rational 的非 explict 构造函数），这也是此混合模式能够成功的原因。

“成功”一词在这一语境下略显尴尬，因为即使代码能够通过编译，也无法进行连接。我们稍后讨论这一问题，先研究一下在 Rational 内声明 operator*时使用的语法。

在类模板内部，模板的名字可作为模板及其参数的缩写，因此在 Rational<T>内部我们就可以将 Rational<T>简写为 Rational。尽管在此示例中这一做法仅让我们少打了几个字母，但是当存在多个参数或长参数名时，这样做可以使代码更简洁、更清晰。我之所以介绍这一点，是因为 operator*声明的参数和返回值类型为 Rational 而不是 Rational<T>。似乎这样声明 opeartor*也是可行的：

```
template<typename T>
class Rational {
public:
  ...
friend
  const Rational<T> operator*(const Rational<T>& lhs,
                              const Rational<T>& rhs);
  ...
};
```

然而还是简写方式更易用（且更常用）。

现在返回到连接的问题。混合模式的代码可以通过编译，这是因为编译器知道我们需要调用那个特定的函数（operator*两个参数的类型：Rational<T>和 Rational<int>），但是这个函数仅仅是**声明**于 Rational 内部，并没有在那里做出**定义**。我们的目标是让类外部的 opeator*模板提供这一定义，但事与愿违。如果我们自己声明了一个函数（我们在 Rational 模板内部就是这样做的），就有

义务对其做出定义。此情况下我们永远也无法提供，这就是为什么连接器无法找到该定义的原因。

最简单的办法就是将 operator*的函数体移入声明中，这似乎可行：

```
template<typename T>
class Rational {
public:
  ...
friend const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),       // 与条目 24 实
                  lhs.denominator() * rhs.denominator()); // 现方法一致
}
};
```

此方法确实可以如期运行：对于 operator*的混合模式调用现在可以编译、连接并运行了，欢呼吧。

通过此技术可以发现一件有趣的事 :使用友元并不能获得对于类的非公共部分的访问权限。为了使所有的实在参数均能够进行类型转换，我们需要一个非成员函数（条目 24 依旧适用）；同时，为了使恰当的函数在恰当的时刻得到实例化，我们需要在类内部声明这些函数。在类内部声明非成员函数的唯一方法就是将其声明为友元。因此我们就这样做了。不合常规码？的确。高效吗？毋庸置疑。

如条目 30 所解释，类内部声明的函数将隐式声明为 inline，包括诸如 opeator*这样的友元函数。通过让 operator*仅仅调用一个定义在类外部的助手函数，即可将此类内联声明带来的影响最小化。本条目的实例中，这样做没有太大意义，因为 operator*本来就只有一行，但是对于更复杂的函数，这个做法很有意义。值得花一点时间研究一下“让友元调用助手函数”的手段。

Rational 是一个模板的事实意味着助手函数同样也是一个模板，因此定义 Rational 的头文件代码应该是这样：

```
template<typename T> class Rational;     // 声明 Rational 模板
```

```
template<typename T>                          // 声明助手模板
const Rational<T> doMultiply(const Rational<T>& lhs,
                             const Rational<T>& rhs);

template<typename T> class Rational {
public:
  ...
friend                                        // 让友元调用助手
  const Rational<T> operator*(const Rational<T>& lhs,
                              const Rational<T>& rhs)
{ return doMultiply(lhs, rhs); }
  ...
};
```

许多编译器从根本上强制将所有的模板定义写入头文件，因此你可能需要在自己的头文件中定义 doMultiply。（如条目 30 所解释，这样的模板不需内联。）代码如下：

```
template<typename T>                    // 如果需要，在头文件中定义助手模板
const Rational<T> doMultiply(const Rational<T>& lhs,
                             const Rational<T>& rhs)
{
  return Rational<T>(lhs.numerator() * rhs.numerator(),
                     lhs.denominator() * rhs.denominator());
}
```

当然，doMultiply 作为一个模板将无法支持混合模式乘法，但是它也无需支持。只有 operator*会调用它，并且 operator*自身支持混合模式操作。本质上，为确保两个 Rational 对象可进行乘法运算，opeartor*函数必须支持任意所需的类型转换，然后将两个对象传入一个以恰当方式实例化的 doMultiply 模板中以进行实际的乘法运算。可谓协同增效。

时刻牢记

- 在编写一个类模板时，如果需要提供一个与该模板相关的、对所有参数支持隐式类型转换的函数，要在类模板内部将其定义为友元。

条目47： 使用 traits 类来描述类型的相关信息

STL 主要由容器、迭代器、算法的模板组成，同时也有一些实用工具模板。其中之一名为 advance。advance 可将一个特定的迭代器移动特定的距离：

```
template<typename IterT, typename DistT> // 将 iter 向前移动 d 个 DisT 单位，
void advance(IterT& iter, DistT d);      // 若 d<0，则将迭代器后移。
```

从概念上讲，advance 等价于 iter += d，但是 advance 并不能那样实现，因为只有随机访问迭代器才支持 += 操作。功能较弱的迭代器类型需要通过迭代进行 d 次 ++ 或 -- 操作来实现 advance。

你不记得 STL 迭代器各个类型了？没关系，我们做一个简单的回顾。迭代器根据所支持的操作分为五种类型。**输入迭代器**仅能向前移动，每次仅能移动一步，仅能读取当前所指的对象，并且对所指对象仅能读取一次。此类迭代器构建于输入文件的读指针；C++库的 istream_iterator 就是此类迭代器的代表。**输出迭代器**与之类似，但其为输出设计：它们仅能向前移动，每次仅能移动一步，仅能改写当前所指对象，且对所指对象仅能改写一次。它们构建于输出文件的写指针；ostream_iterator 是此类迭代器的一个典型。这两类迭代器功能最弱。因为输入输出迭代器仅能向前移动，且对所指对象仅能做出一次读写操作，它们只适用于“一趟算法”。

向前迭代器是一类功能更强的迭代器。除了拥有输入和输出迭代器的所有功能以外，它们还可以对所指的对象做出超过一次的读写操作。这就使它们可用于多趟算法。STL 没有提供单链表，但一些库中有提供（一般称为 slist），此类容器的迭代器就是向前迭代器。TR1 中哈希容器（参见条目 54）的迭代器也属于向前迭代器类型。

双向迭代器为向前迭代器添加了向后移动的功能。STL 中 list、set、multiset、map 以及 multimap 的迭代器就属于这一类型。

功能最强大的迭代器类型是**随机访问迭代器**。此类型的迭代器为双向迭代器添加了进行“迭代器算数”的能力，也就是说，可在恒定时间内向前或向后跳跃任意的距离。迭代器算数与指针算数类似，这并不稀奇，因为随机访问迭代器就是通过内建指针

实现的，而内建指针也可像随机访问迭代器一样使用。vector、deque 和 string 的迭代器就是随机访问迭代器。

对于上述五类迭代器，C++标准库均内置了一个“标记结构体”以便识别：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

这些结构体之间的继承关系是合法的“A 是一个 B”关系（参见条目 32）：“所有的向前迭代器都是输入迭代器”等说法均成立。此继承结构的作用稍后解释。

先回到 advance。对于迭代器不同的功能级别，实现 advance 的看似可行的途径之一就是使用“最小公分母”策略，使用一个循环来迭代增减操作。但是这一方案时间复杂度为 O(*n*)。而随机访问迭代器应支持恒定时间的迭代器运算，我们不妨顺水推舟。

我们真正想做的是按下面（大体的）方案来实现 advance：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d) {
  if (iter是一个随机访问迭代器) {
    iter += d;                              // 对随机访问迭代器进行迭代器算数
  }
  else {
    if (d >= 0) { while (d--) ++iter; }   // 对其他迭代器类型使用
    else { while (d++) --iter; }          // 对++或--的迭代调用
  }
}
```

此方案需要事先知道 iter 是否是随机访问迭代器，继而需要知道 IterT（iter 的类型）是否是随机访问迭代器类型。换句话说，我们需要取得类型的一些信息。这就是 traits 的能力：使你能够在编译时取得类型信息。

traits 不是 C++的关键词，也不是事先定义的结构体；它是一项技术，是 C++程序

员遵循的一个惯例。对此技术，人们期望对内建类型可以像用户定义类型那样正常工作。比如，如果 advance 通过一个指针（就像 const char *）和一个 int 调用，那么它必须正常工作，但是这就意味着 traits 技术必须适用于诸如指针的内建类型。

“traits 必须适用于内建类型”就意味着类型内嵌信息的方式不能奏效，因为我们无法在指针内部嵌入信息。于是一个类型的 traits 信息就必须位于类型的外部。标准的做法是将其置于一个模板以及一个或多个该模板的特化版本中。对于迭代器，标准库中的模板名为 iterator_traits：

```
template<typename IterT>          // 用于保存迭代器类型信息的模板
struct iterator_traits;
```

如你所见，iterator_traits 是一个结构体。依照惯例，traits 总会实现为结构体。另一个惯例就是用于实现 traits 的结构体通常叫做“traits **类**”（这可不是我编的）。

iterator_traits 这样运行：对于每个类型 IterT，结构体 iterator_traits<IterT> 中都会声明一个名为 iterator_category 的 typedef。由它来识别 IterT 迭代器种类。

iterator_traits 的实现分为两部分。第一，它强制要求所有的用户定义迭代器类型必须包含一个内嵌的名为 iterator_category 的 typedef 以识别恰当的标记结构体。比如 deque 的迭代器是随机访问的，因此 deque 迭代器的类会类似于：

```
template < ... >                  // 省略模板参数
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  };
  ...
};
```

而 list 的迭代器是双向的，因此迭代器的类是这样的：

```
template < ... >
class list {
public:
  class iterator {
  public:
    typedef bidirectional_iterator_tag iterator_category;
    ...
  };
  ...
};
```

iterator_traits 只是机械地重复了迭代器类的内嵌 typedef：

```
// IterT 说什么类型，iterator_category 就是什么类型；
// "typedef typename"的用法参见条目 42
template<typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator_category iterator_category;
  ...
};
```

这对用户定义类型可行，但当迭代器是指针时毫无意义，这是因为不存在"指针内嵌 typedef"的概念。iterator_traits 实现的第二部分将解决迭代器是指针的情形。

为支持此类迭代器，iterator_traits 为指针类型提供了一个偏特化的模板。由于指针运行方式与随机访问迭代器类似，因此 iterator_traits 为其选定了这一类型：

```
template<typename T>                // 对内建指针进行模板偏特化
struct iterator_traits<T*>
{
  typedef random_access_iterator_tag iterator_category;
  ...
};
```

现在你应该知道如何设计和实现一个 traits 类了：

- 对所需的类型明确必要信息。（正如，迭代器的种类）
- 为该信息选定一个名字。（正如 iterator_category）
- 提供一个模板和一组实例化版本（正如 iterator_traits），其中应包含需要支持类型的信息。

对于 iterator_traits（实际是 std::iterator_traits，因为它是 C++标准库的一员），我们可以将 advance 的伪代码做出如下改进：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
      typeid(std::random_access_iterator_tag))
  ...
}
```

虽然看上去按部就班，可却暗藏杀机。首先，这将导致编译问题（我们将在条目 48 中讨论）；现在需要考虑的是一个更基础的问题。由于 IterT 的类型在编译时已知，因此 iterator_traits<IterT>::iterator_category 也可在编译时确定。而 if 语句是在运行时执行的（除非编译器足够聪明自行优化）。在编译时能搞定的事为什么要留到运行时呢？这样着实会浪费时间，也会使可执行文件膨胀。

我们真正需要的是一个在编译时执行的类型判断条件结构（也就是一个 if…else 语句）。C++有办法达成这一行为。即重载。

在重载函数 f 时，你为不同的重载版本指定了不同的参数类型。当你调用 f 时，编译器会基于你传入的实际参数挑选出最佳重载版本。编译器大致会这样说："如果这一重载版本对于正在传入的参数是最佳匹配，就调用此 f；如果另一个重载版本是最佳匹配，就调用另一个；如果第三个是最佳，就调用第三个，以此类推。"看到了？这是一个编译时用来确认类型的条件结构。为了使 advance 按我们期望的方式运行，我们需要做的就是创建多个重载版本的函数，并将 advance 的内在结构包含在内，声明每个版本时各取一个不同类型的 iterator_category 对象作为参数。我将这系列函数命名为 doAdvance：

```
template<typename IterT, typename DistT>          // 本实现用于
```

```
void doAdvance(IterT& iter, DistT d,                // 随机访问迭代器
               std::random_access_iterator_tag)
{
  iter += d;
}

template<typename IterT, typename DistT>            // 本实现用于
void doAdvance(IterT& iter, DistT d,                // 双向迭代器
               std::bidirectional_iterator_tag)
{
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter; }
}

template<typename IterT, typename DistT>            // 本实现用于
void doAdvance(IterT& iter, DistT d,                // 输入迭代器
               std::input_iterator_tag)
{
  if (d < 0) {
    throw std::out_of_range("Negative distance");  // 参见下文
  }
  while (d--) ++iter;
}
```

由于 forward_iterator_tag 继承自 input_iterator_tag，input_iterator_tag 的 doAdvance 版本同样适用于向前迭代器。这也就是不同 iterator_tag 结构体之间使用继承结构的积极作用所在。（事实上，这是**所有**公有继承的积极作用之一：让对于基类类型使用的代码同样适用于派生类类型。）

advance 的说明书中载明了：随机访问和双向迭代器的移动距离即可为正也可为负，但是如果你试图向后移动一个向前或输入迭代器则会带来未定义行为。我调查过一些实现，它们只是简单的假设 d 为非负，这样如果传入一个负的移动距离将会导致一个非常漫长的、“倒计时”至零的循环。上文的代码中我改用抛出异常的方案。两种实现都是合法的。这便是未定义行为的咒语，你**无法预知**下一秒会发生什么。

所有的 advance 都需要做出修改以便调用 doAdvance 的多个重载版本，多传入

一个对象用于保存恰当的迭代器种类类型，以使编译器可以通过重载解析来调用恰当的实现：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  doAdvance(                          // 调用那个适用于 iter 的迭代器类型的
    iter, d,                          // doAdvance 版本
    typename std::iterator_traits<IterT>::iterator_category()
  );
}
```

现在我们可以总结一下 traits 类的用法：

- 创建一组重载的“工作”函数或函数模板（也就是 doAdvance），它们各自有不同的 traits 参数，依照传入的信息来实现每个函数。
- 创建一个“主”函数或函数模板（也就是 advance），它们通过传递 traits 类提供的信息来调用“工作函数”。

traits 技术在标准库中的到了广泛应用。当然，在 iterator_category 以外，iterator_traits 还提供了迭代器其他四项信息（最有用的是 value_type，条目 42 中有个示例说明了其用法）。还有保存字符类型信息的 char_traits、保存了数字类型信息的 numeric_limits，比如可表示的最大值、最小值，等等。（numeric_limits 的名字略显以外，因为 traits 类更普遍的命名传统是以“traits”为结尾，但是 numeric_limits 名字更直观，因此我们使用这个名字。）

TR1 进入了众多新的提供类型信息的 traits 类，包括 is_fundamental<T>（T 是否为内建类型）、is_array<T>（T 是否是一个数组）、is_base_of<T1, T2>（T1 是否与 T2 相同或是其基类）。TR1 总共为标准 C++添加进了 50 余个 traits 类。

时刻牢记

- traits 类在编译时就明确了类型信息。他们通过模板和模板特化版本实现。

- traits 类配合重载功能，使得对于类型的 `if…else` 测试可在编译时进行。

条目48： 留意模板元编程

模板元编程（template metaprogramming，简称 TMP）是编写“在编译时运行的基于模板的 C++程序”的过程。请考虑：模板元程序是用 C++编写的在 **C++编译器内部**运行的程序。当 TMP 程序运行结束时，它的输出结果（由模板实例化出的一段 C++代码）像平时一样得到编译。

如果模板元编程只是波澜不惊的一项技术，那么只是因为你没有深入考虑罢了。

C++并非为模板元编程设计，但是自上世纪九十年代初期人们发现 TMP 至今，它的用途已被证实，C++语言本身和标准库很可能会针对 TMP 进行扩展，届时其用法将更简便。是的，TMP 是被发现的，并不是人们的发明。基于 TMP 的特性是在 C++加入模板功能后引入的。只是一些黑客炫技的手段。

TMP 有两大能力。其一，它可以使一些非常困难或原本不可能实现的事情变得简单。其二，由于模板元程序在 C++编译期间运行，因此可以把工作由运行时提前至编译时。结果之一就是通常在运行时才能识别的错误可以在编译时发现了。另一个是使用 TMP 的 C++程序在各个方面都更高效：可执行程序更小、运行时间更短、所需内存更少。（然而，把工作由运行时提前至编译时也会让编译时间更长。TMP 程序的编译时间比非 TMP 的版本要长**很多**。）

请观察下面 STL 中 `advance` 的伪代码（参见条目 47。你需要重新阅读该条目，因为本条目中我将假设你了解这段伪代码的细节。）和条目 47 中一样，我将伪代码部分加粗：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter 是一个随机访问迭代器) {
    iter += d;                          // 为随机访问迭代器使用迭代器算数
  }
  else {                                // 对其他类型迭代器使用++或--的迭代调用
    if (d >= 0) { while (d--) ++iter; }
```

```
    else { while (d++) --iter; }
  }
}
```

我们可以使用 typeid 让这段伪代码变为真实代码。这将使此问题以一个“正常的”C++方法（即在运行时完成所有工作）实现。

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
      typeid(std::random_access_iterator_tag)) {
    iter += d;                          // 为随机访问迭代器使用迭代器算数
  }
  else {                                // 对其他类型迭代器使用++或--的迭代调用
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

条目 47 中记载了，基于 typeid 的方法比 traits 方法效率更低，这是因为此方法（1）在运行时测试类型，而不是编译时，并且（2）运行时类型测试代码必须出现在可执行程序中。事实上，本示例展示了 TMP 相对“正常的”C++程序而言是怎样做到更高效的， 因为 traits 方法**就是** TMP。请记住，traits 使得判断类型的 if…else 语句可以在编译时进行。

我前面说过 TMP 中的某些东西比“正常的”C++更简单，advance 就是一个示例。条目 47 中提到了 advance 基于 typeid 的实现会导致编译问题，以下是出现问题的示例：

```
std::list<int>::iterator iter;
...
advance(iter, 10);                      // 迭代器前移 10 个元素。
                                        // 使用上文的迭代器无法通过编译。
```

请考虑上文的调用将会生成什么版本的 advance。在为 IterT 和 DistT 替换了 iter 和 10 的类型后，我们得到了以下代码：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (
    typeid(
      std::iterator_traits<std::list<int>::iterator>::iterator_category)
    == typeid(std::random_access_iterator_tag) ) {
    iter += d;                          // 错误！无法通过编译
  }
  else {
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

问题出现在加粗的一行，其中使用了+=。在此情况下，我们试图对`list<int>::iterator`运行+=，但`list<int>::iterator`是一个双向迭代器，因此不支持+=。只有随机访问迭代器支持+=。现在我们知道我们的程序永远不会运行那一行，因为`typeid`对于`list<int>::iterator`的测试总会失败，但是编译器更倾向于确认所有源代码均合法，甚至是不运行的部分，而"`iter += d`"在`iter`不是随机访问迭代器的情况下是非法的。基于traits的TMP方案则不同，不同类型的代码会分散至不同的函数中，每个函数只使用为特定类型编写的适当的操作。

TMP是图灵完备的，这就意味着它可以胜任所有的计算。使用TMP可以声明变量、运行循环、编写和调用函数，等等。但是这类结构与它们所对应的"正常"的C++版本有着天壤之别。举例说，条目47中展示了TMP是如何通过模板和模板特化描述`if…else`条件的。但是这是汇编级的TMP。TMP的库（比如Boost中的MPL，参见条目55）提供了更高级的语法，但是你依然不会与"正常"的C++搞混。

让我们换个角度领略一下TMP的工作方式，以循环为例。TMP并没有真正的循环结构，因此循环的效果是通过递归达成的。（如果你不熟悉递归，那么你在进行TMP探险之前应先了解它。C++主要是函数式语言，而递归与函数式语言的关系就像电视和美国流行文化之间的关系：二者形影不离。）然而，甚至递归都不是正常的类型，因为TMP循环并不包含递归函数调用，而是包含递归**模板实例化**。

TMP 的“hello world”程序是在编译时进行阶乘运算。这个程序并不让人兴奋，但也不“hello world”，作为语言特性的介绍很有帮助。TMP 阶乘运算展示了通过递归模板实例化进行循环。同时也展示了 TMP 中创建和使用变量的一种方法。请观察：

```
template<unsigned n>                    // 一般情况：Factorial<n>的值是 n 倍的
struct Factorial {                      // Factorial<n-1>
  enum { value = n * Factorial<n-1>::value };
};

template<>                              // 特殊情况：Factorial<0>的值是 1
struct Factorial<0> {
  enum { value = 1 };
};
```

有了这个模板元程序（确实只有一个模板元程序 `Factorial`），你可以通过调用 `Factorial<n>::value` 得到 *n*!。

当模板实例 `Factorial<n>`引用模板实例 `Factorial<n-1>`时，循环代码出现了。就像所有好的递归程序，这里存在一个特殊情况可以导致递归终止。即特化的 `Factorial<0>`。

`Factorial` 模板的实例均为结构体，并且每个结构体均使用 enum 黑客手段（参见条目 2）来声明名为 `value` 的 TMP 变量。`value` 保存着当前阶乘的计算值。如果 TMP 存在真正的循环结构，那么 `value` 则会在每次循环时均更新一次。由于 TMP 使用递归模板实例化取代了循环，每个实例都得到了自己的 `value` 副本，并且每个副本均拥有与“循环”中匹配的恰当数值。

你可以这样使用 `Factorial`：

```
int main() {
  std::cout << Factorial<5>::value;       // 打印 120
  std::cout << Factorial<10>::value;      // 打印 3628800
}
```

如果你觉得这一方案酷到骨子里，那么你就具备了做一个模板元编程程序员的条件。如果模板、特化版本、递归实例化、enum 黑客手段以及编写类似 `Factorial<n-1>::value` 的程序让你毛骨悚然，那么，你还是做一名普通 C++

程序员吧。

当然，`Factorial` 向我们展示的 TMP 用法，就好比“hello world”之于传统编程语言那样初级。为了探究 TMP 的实际价值，有必要更好地了解一下它可以达成的功能。以下是三个示例：

- **保证度量衡单位的正确性。**在科学和工程程序中，正确组合度量衡单位（比如质量、距离、时间，等等。）是一件至关重要的事情。比如，把一个表示质量的变量赋值给一个表示速度的变量就是错误的，但是用一个距离变量除以一个时间变量并将结果赋值给一个速度变量又是正确的。使用 TMP 可以（在编译时）确保程序所有度量衡单位组合的正确性。（这是 TMP 应用于早期错误识别的一个示例。）TMP 的这一用途还有一个有趣的方面：它支持分数指数的单位。为了使编译器确认这一数值，该分数应该在编译时可被约分，比如，$time^{\frac{1}{2}}$ 与 $time^{\frac{4}{8}}$ 等价。

- **优化矩阵操作。**条目 21 中解释了包括 operator*等此类函数应返回新的对象，同时条目 44 中引入了 SquareMatrix 类，因此请考虑下面的代码：

```
typedef SquareMatrix<double, 10000> BigMatrix;
BigMatrix m1, m2, m3, m4, m5;                    // 创建矩阵
...                                              // 并赋值

BigMatrix result = m1 * m2 * m3 * m4 * m5;       // 计算矩阵的乘积
```

  通过“正常”的方法计算 result 将会带来四个临时矩阵的开销，每次调用 `operator*`的结果都会使用一个。并且，独立的乘法将会引入一系列四次针对矩阵元素的循环。使用与 TMP 相关的一种更高级的模板技术——“表达式模板”，可以消除这些临时矩阵以及合并循环，所有一切都不需要上面的客户端代码做出任何语法修改。这样的程序占用内存更小，且增速显著。

- **创造定制化的设计模式实现。**诸如策略模式、观察者模式、访问者模式等设计模式均可以多种方法实现。一个基于 TMP 的技术被称为“基于原则设计”，它让你能够创建表示独立设计方案选项（即“原则”）的模板，而不同的设计方案可以任意组合，从而产生具有特定行为的模式实现。举例说，此技术用于让某些

模板实现智能指针的行为原则，从而（在编译时）生成上百种不同的智能指针类型。模板元编程技术也是产生式编程的一项基础，这已经远远超出了设计模式和智能指针这些编程副产品的领域了。

TMP 并不对于所有人都适用。它语法刁钻，工具支持脆弱。（别和我提模板元编程的调试器。）由于它是一个不久前“偶然”发现的语言特性，TMP 程序尚无统一规律可循。然而，通过把工作由运行时提前至编译时可以显著改善程序效率，同时 TMP 也可描述那些在运行时难以实现甚至无法实现的行为，这一点也是振奋人心的。

TMP 的支持日新月异。C++的下一版本很有可能对其做出显式支持，TR1 已经在这样做了（参见条目 54）。关于这一主题的书目即将面世，网络中 TMP 的信息也是遍地开花。尽管 TMP 永远不会成为主流，但是对于一些程序员而言，尤其是库开发者，必将变得举足轻重。

时刻牢记

- 模板元编程可以将运行时的工作提前至编译时进行，这样可以使错误提早发现，并提高运行性能。
- TMP 可以用于生成基于策略选择组合的个性化代码，也可以用于避免为特定的不恰当的类型生成代码。

第八章. 定制 new 和 delete

当今的语言都在大肆宣传内建的垃圾收集支持（比如 Java 和.NET），使用 C++手动管理内存看上去太过时了。但是许多开发者在开发那些要求较高的系统程序时均会选用 C++**正是因为**它支持手动管理内存。此类开发者会研究软件的内存使用特征，他们会量身定制内存分配和释放规则，从而使他们正在构建的系统尽可能取得最优性能（包括时间和空间）。

做到这一点需要对 C++内存管理机制的行为有一定的了解，即本章的主题。舞台上有两位主角：分配规则（operator new）和释放规则（operator delete），和一位配角：new 处理函数（new-handler，在 operator new 不能满足内存需求时可供调用的函数）。

多线程环境中的内存管理会面对单线程系统中不存在的挑战，因为堆和 new 处理函数都是可修改的全局资源，它们都遵循竞争条件，不适于多线程系统。本章许多条目都提到了可修改的静态数据，这些东西对于线程敏感的程序员可谓“九阴真经”。如果没有恰当的同步机制，使用无锁算法、防止并发访问的“精心”设计，内存调用规则均有可能“走火入魔”，即导致令人困惑的行为，或损坏由堆管理的数据结构。与其在今后反复强调这一危险，不如现在做出提醒并在本章始终牢记在心。

另外需要留意的是 operator new 和 operator delete 只适用于单个对象。数组的内存是通过 operator new[]和 operator delete[]进行分配和释放的。（请注意两种情况的差别在于函数名中的“[]”。）除非我额外说明，本章中所写的关于 operator new 和 operator delete 的内容对于 operator new[]和 operator delete[]同样适用。

最后，请注意 STL 容器的内存是通过容器的分配器对象来管理的，而非直接使用 new 和 delete，鉴于此，本章不涉及 STL 分配器。

条目49： 理解 new 处理函数的行为

当 operator new 无法满足内存分配请求时，它将抛出一个异常。很久以前，它返回一个空指针，至今一些古老的编译器仍在这样做。你仍可以（在某种意义上）

利用这一古老的行为，这一点到本条目最后再讨论。

在 operator new 对于一次不满足条件的内存请求抛出异常之前，它将首先访问一个名为 **new 处理函数**（**new-handler**）的错误处理函数。（这也不是完全正确。operator new 真正做的事更复杂些。条目 51 将提供细节。）客户通过调用 set_new_handler 来指明“处理内存不足的函数”，<new>中定义了这个标准库函数：

```
namespace std {
  typedef void (*new_handler)();
  new_handler set_new_handler(new_handler p) throw();
}
```

如你所见，new_handler 是一个 typedef，它是一个无参数无返回值的函数指针，set_new_handler 是一个以 new_handler 为参数并返回一个 new_handler 的函数。（set_new_handler 末尾声明的“throw()”是一个异常明细表。它大体是说此函数不会抛出任何异常，而真相另有玄机。细节参见条目 29。）

operator new 不能分配足够的内存时会调用一个函数，set_new_handler 将指向此函数的指针作为参数，以在其调用前用于完成此工作的函数的指针作为返回值。

你可以这样使用 set_new_handler：

```
// operator new 无法分配足够内存时调用的函数：
void outOfMem()
{
  std::cerr << "无法申请足够内存\n";
  std::abort();
}

int main()
{
  std::set_new_handler(outOfMem);
  int *pBigDataArray = new int[100000000L];
...
}
```

如果 operator new 无法为 1 亿个整数分配分配足够空间，那么程序将调用 outOfMem，并且在发布一条出错信息后终止运行。(同时请考虑，如果在向 cerr 写入出错信息时内存必须是动态分配的，将会发生什么。)

若 operator new 无法满足内存请求，它将反复调用 new 处理函数直至**可以**寻找到足够内存为止。导致反复调用的代码将在条目 51 中展示。此处的更抽象的描述已经足以得出一个结论——“要将 new 处理函数设计好必须满足以下条件之一”：

- **分配足够内存。**这使下次 operator new 的分配尝试可能成功。实现此策略的方法之一是：在程序开始时分配一大块内存，然后在 new 处理函数首次调用时释放以备使用。
- **安装另一个 new 处理函数。**如果当前的 new 处理函数无法分配更多内存，可能是它知道另一个 new 处理函数可以做到。如果是这样，当前的 new 处理函数就可以在它的位置安装另一个 new 处理函数（通过调用 set_new_handler）。下次 operator new 在调用 new 处理函数时，它将得到最近安装的一个。(B 计划是：修改 new 处理函数自身的行为，以使其下次调用时可以做不同的工作。一种实现方法是：让 new 处理函数来修改可以影响其行为的那些 static 的、特定名字空间内的，或全局的数据。)
- **卸载 new 处理函数。**也就是将空指针传递给 set_new_handler。没有安装 new 处理函数的情况下，operator new 将会在内存分配失败时抛出一个异常。
- **抛出异常。**类型为 bad_alloc 或其派生类型。由于这样的异常将不会被 operator new 捕获，因此它们也不会传播至内存需求的源头。
- **不要返回。**一般情况下通过调用 abort 或 exit 实现。

这些选择让你可以更灵活地实现 new 处理函数。

分配对象从属于不同的类，这样会导致各种内存分配失败问题，有时，你期望分别处理好这些问题：

```
class X {
public:
  static void outOfMemory();
```

```
  ...
};

class Y {
public:
  static void outOfMemory();
  ...
};

X* p1 = new X;                    // 如果分配失败则调用 X::outOfMemory
Y* p2 = new Y;                    // 如果分配失败则调用 Y::outOfMemory
```

C++不支持特定类的 new 处理函数，也并不需要。你可以自行实现这个行为。你可以为每个类提供自有的 set_new_handler 和 operator new 版本。每个类的 set_new_handler 允许客户为其定制 new 处理函数（就如同标准的 set_new_handler 允许客户定制全局 new 处理函数）。特定类的 operator new 保证了在为类对象分配内存时，该类的 new 处理函数将取代全局 new 处理函数。

假设你想为 Widget 类处理内存分配失败问题。你应在 operator new 无法为 Widget 对象分配足够内存时保持对所调函数的追踪，因此你应该声明一个静态的 new_handler 类型的成员来指向 Widget 的 new 处理函数：

```
class Widget {
public:
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);
private:
  static std::new_handler currentHandler;
};
```

静态类成员必须定义在类定义的外部（除非他们是 const 并且是整数，参见条目 2），因此：

```
std::new_handler Widget::currentHandler = 0;   // 在类定义文件中初始化为 null
```

Widget 的 set_new_handler 函数将保存传入的指针，并返回调用前保存的指针（无论指针类型）。这正是标准 set_new_handler 版本所做的：

```
std::new_handler Widget::set_new_handler(std::new_handler p) throw()
{
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
```

最终，Widget 的 operator new 将做下面的事情：

1. 使用 Widget 的错误处理函数来调用标准 set_new_handler。这将使得 Widget 的 new 处理函数按全局 new 处理函数的形式来安装。
2. 调用全局 operator new 来执行实际的内存分配工作。如果分配失败，全局 operator new 就会调用 Widget 的 new 处理函数，这是因为该函数刚安装为全局 new 处理函数。如果全局 opeartor new 最终仍无法分配内存，那么将抛出一个 bad_alloc 异常。在此情况下，Widget 的 operator new 必须还原原始的全局 new 处理函数，然后传播该异常。为了确保原始的 operator new 总能够得到恢复，Widget 会将全局 new 处理函数当做一个资源，并且会遵循条目 13 中的建议使用资源管理类来防止资源泄露。
3. 如果全局的 operator new 可以为 Widget 对象分配足够内存，那么 Widget 的 operator new 将返回一个指向所分配内存的指针。管理全局 new 处理函数的对象将由析构函数自动将全局 new 处理函数还原为 Widget 调用 operator new 之前的状态。

以下我们用 C++代码来描述一遍。资源处理类包括在构造时申请资源和析构时释放资源这样的基础 RAII 操作（参见条目 13），我们从资源处理类开始：

```
class NewHandlerHolder {
public:
  explicit NewHandlerHolder(std::new_handler nh)
  : handler(nh) {}                              // 申请当前 new 处理函数

  ~NewHandlerHolder()                           // 释放当前 new 处理函数
  { std::set_new_handler(handler); }
```

```
private:
  std::new_handler handler;                    // 请记住

  NewHandlerHolder(const NewHandlerHolder&); // 防止复制（参见条目 14）
  NewHandlerHolder& operator=(const NewHandlerHolder&);
};
```

这使得实现 Widget 的 operator new 变得十分简单：

```
void* Widget::operator new(std::size_t size) throw(std::bad_alloc)
{
  NewHandlerHolder h(std::set_new_handler(currentHandler));
                                    // 安装 Widget 的 new 处理函数
  return ::operator new(size);      // 分配内存，或抛出异常
}                                   // 还原全局 new 处理函数
```

Widget 的客户可以这样使用它的 new 处理功能：

```
void outOfMem();                    // 函数声明，用于在 Widget 对象
                                    // 内存分配失败时调用。

Widget::set_new_handler(outOfMem); // 将 outOfMem 设置为 Widget 的 new
                                    // 处理函数

Widget *pw1 = new Widget;           // 如果内存分配失败，调用 outOfMem

std::string *ps = new std::string; // 如果内存分配失败，调用全局 new
                                    // 处理函数（如果存在）

Widget::set_new_handler(0);         // 将针对 Widget 的 new 处理函数归零
                                    //  （也就是 null）

Widget *pw2 = new Widget;           // 如果内存分配失败，立即抛出异常。
                                    //  （这里不存在 Widget 类的 new 处理函数）
```

无论什么类，实现这一方案的代码都是一致的，因此有理由要求代码可以在其他场合复用。一个简单的方法即：创建“mixin 风格”的基类，或者说允许派生类继承单一

特定的功能的基类。也就是此情况下的“设置特定类 new 处理函数。”下面我们把基类转换为模板，对于类的数据每个派生类将生成各不相同的拷贝。

本设计的基类部分让所有派生类均继承所需的 set_new_handler 和 operator new 函数，而本设计的模板部分保证每个派生类都拥有一个不同的 currentHandler 数据成员。这可能听上去有一些复杂，但是这里的代码看上去似曾相识。事实上，唯一真正的不同就是：此时的代码对于任何需要它的类都可用了。

```
template<typename T>                // “mixin 风格”基类，
class NewHandlerSupport {           // 提供对特定类的 set_new_handler 支持
public:
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);
  ...                               // operator new 的其他版本，参见条目 52

private:
  static std::new_handler currentHandler;
};

template<typename T>
std::new_handler
NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}

template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
  throw(std::bad_alloc)
{
  NewHandlerHolder h(std::set_new_handler(currentHandler));
  return ::operator new(size);
}
```

```
template<typename T>                    // 将每个 currentHandler 初始化为 null
std::new_handler NewHandlerSupport<T>::currentHandler = 0;
```

有了这个类模板，为 Widget 添加 set_new_handler 支持就非常简单了：Widget 只需继承自 NewHandlerSupport<Widget>。(这看上去有些奇怪，但是我将在下文中对于所发生的事情做更细致的讨论。)

```
class Widget: public NewHandlerSupport<Widget> {
  ...                                   // 与以前一样，但是没有 set_new_handler
                                        // 或 operator new 的声明
}
```

这就是 Widget 提供支持特定类的 set_new_handler 所要做的一切。

但是你可能对与 Widget 继承自 NewHandlerSupport<Widget>一事耿耿于怀。如果真的这样，那么当你发现了 NewHandlerSupport 模板永远也不会使用类型参数 T，你会愈发担心。其实没有必要。我们需要的只是为每个继承自 NewHandlerSupport 的类提供 NewHandlerSupport (更具体地说是它的静态数据成员 currentHandler) 的一个不同的副本。模板参数 T 恰可区别不同的派生类。模板机制本身就可以在 NewHandlerSupport 实例化时为当前的 T 自动生成一个 currentHandler 的副本。

对于让 Widget 继承自一个以 Widget 为类型参数的模板化基类一事，如果你有些头晕脑胀，不要灰心。每个人一开始都是这样的感觉。然而，事实证明它是一项实用技术，它拥有一个名字——“**奇异递归模板模式**”(curiously recurring template pattern，CRTP)，这个名字能反映出人们初次见到此模式时都会觉得它很奇怪。的确很“奇异”。

为这件事我曾发表过一篇文章建议使用“为我做事”这个名字可能更好些，因为 Widget 继承自 NewHandlerSupport<Widget>实际上意味着“我是 Widget，我想为 Widget 继承 NewHandlerSupport 类。”没人采纳我的建议（甚至我自己也没有)，但是将 CRTP 理解为“为我做事”的一种方式可以让你更好地理解模板化继承。

对于所有需要特定的 new 处理函数的类而言，类似 NewHandlerSupport 这样的

模板让添加 new 处理函数变得十分简单。然而 mixin 风格的继承必定导致多重继承，在你选择这条路之前建议你阅读一下条目 40。

1993 年前，C++在无法分配足够内存时需要 operator new 返回空值。现如今细化的 operator new 则抛出一个 bad_alloc 异常，但是许多 C++代码编写于编译器开始支持这一改进的规范之前。由于 C++标准委员会并不希望放弃"测试空值"的代码库，因此他们引入了 operator new 的其他形式，从而提供了传统的"错误产生空值"行为。它们被称为"零异常"（"nothrow"）形式，一定程度是因为它们在使用 new 时引入了 nothrow 对象（定义于<new>头文件中）：

```
class Widget { ... };

Widget *pw1 = new Widget;          // 如果分配失败则抛出 bad_alloc
if (pw1 == 0) ...                  // 测试必定失败

Widget *pw2 = new (std::nothrow) Widget;
                                   // 如果 Widget 分配失败则返回 0
if (pw2 == 0) ...                  // 测试可能成功
```

nothrow 的 new 乍看去强制禁止异常，但事实上它会略宽松些。在"new (std::nothrow) Widget"语句中会发生两件事。第一，在 Widget 对象分配足够内存时会调用 operator new 的零异常版本。如果分配失败，则 opeartor new 将像展示的那样返回一个空指针。然而如果分配成功了，将会调用 Widget 的构造函数，此时一切前途未卜。Widget 的构造函数可能为所欲为。它可能会自己 new 一些内存，如果它这样做了，这里的 new 是没有零异常强制保证的。尽管 new (std::nothrow) Widget 中的 operator new 调用不会抛异常，Widget 的构造函数也有可能抛，这是第二。如果真的抛了，异常将一如既往得到广播。结论？就是使用 nothrow 的 new 只能保证当前的 operator new 不抛异常，而无法保证"new (std::nothrow) Widget"这样的语句永远不会产生异常。你十有八九永远也不会用到 nothrow 的 new。

不管你是使用"正常"的（也就是抛异常的）new 还是什么炫技的零异常版本，理解 new 处理函数的行为还是十分有必要的，因为两种情况都会用到。

时刻牢记

- set_new_handler 允许你在内存分配请求无法满足时提供定制函数。
- 零异常的 new 只有有限的实用价值，因为它仅适用于内存分配，与之相关的构造函数调用仍会抛出异常。

条目50： 要清楚什么时候应使用 new 和 delete 的替代方案

这一刻让我们回归本真。为什么有人一开始就要为编译器提供的 operator new 和 operator delete 寻找替代方案啊？以下是三个最普遍的理由：

- **甄别使用错误。**用 delete 释放 new 出的内存失败时，将导致内存泄漏。对 new 出的内存使用 delete 释放多于一次时，将导致未定义行为。如果用 operator new 保存一个内存分配表，并且 operator delete 在释放后从表中将地址移除，那么甄别此类使用错误的工作就简单多了。类似的，各种编程错误可能会导致数据越界（在分配好的内存块的尾部之后写入数据）或反向越界（在分配好的内存块的头部之前写入数据）。定制 operator new 可以分配一块额外的空间，以在客户可见的内存之前或之后存放约定好的代码模式（即“签名”）。operator delete 可以检查该签名是否仍完好保存。如果它们遭受破坏，就意味着已分配的内存块在生命周期中发生了数据越界或反向越界，operator delete 可以记录这一事故，以及出错指针的值。

- **改进性能。**编译器提供的 operator new 和 operator delete 为一般应用场景设计。它们应该既能满足类似 web 服务器那样的长期程序，又能满足仅执行一秒的短期程序。无论大块的、小块的内存，还是二者混合的情况，它们都应该能够成功处理各种请求。无论是存在于整个程序运行过程中少量内存块的动态分配，还是大量连续不断的短期对象分配和释放，它们都应该能够完成分配工作。它们需要关心堆碎片问题，如果忽视之，最终会导致程序无法完成大块内存的请求，即使在诸多的小块内存之间分布着足够的内存空间。

  看到内存管理的诸多要求，编译器提供的 operator new 和 operator delete 采取折中策略就不难理解了。对于所有人而言它们运行得还算可以，

但是没有为任何人做出优化。如果你对于当前程序动态内存使用模式有较好的理解，那么你经常会遇到“自定义的 operator new 和 opeartor delete 版本比默认版本优秀”的情况。我说“优秀”是指运行速度更快（有时会十分显著）且占用内存更少（节约近 50%）。对于一些（并非所有）程序，将原始的 new 和 delete 替换为自定义版本能够很轻易带来显著的性能提升。

- **收集内存使用统计数据。**在开始为你的软件编写自定义的 new 和 delete 之前，收集使用动态内存的信息是明智的做法。分配内存块的大小、生命期限是怎样分布的？这些内存块的分配和释放顺序是 FIFO（“first In, first out”，“ 先入先出”），还是 LIFO（“last in, first out”，“后入先出”），亦或某些更接近于随机的顺序？内存使用模式是否经常变化（举例说，你的软件在不同运行阶段是否存在不同的分配/释放模式）？动态内存在任意时刻的最大用量是多少（也就是“高水位线”）？让自定义版本的 operator new 和 operator delete 收集这类信息如探囊取物。

理论上讲，编写自定义的 operator new 非常简单。比如，以下是一个全局 operator new 的改善版本，它优化了正反两向的越界检查。代码中还存在不少细节错误，我们稍后再提。

```
static const int signature = 0xDEADBEEF;
typedef unsigned char Byte;

// 代码有瑕疵，参见下文。
void* operator new(std::size_t size) throw(std::bad_alloc)
{
  using namespace std;
  size_t realSize = size + 2 * sizeof(int);
                                    // 增加请求的空间，以容纳 2 个签名。
  void *pMem = malloc(realSize);    // 调用 malloc 来取得实际内存
  if (!pMem) throw bad_alloc();

  // 在内存的头部和尾部写入签名
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem) +
    realSize - sizeof(int))) = signature;
```

```
  // 返回一个指针：指向紧挨头部签名后方的内存
  return static_cast<Byte*>(pMem) + sizeof(int);
}
```

此 operator new 最大的缺点在于其没有遵循 C++中 new 函数编写的惯例。比如，条目 51 解释了，所有 operator new 都应该包含一个调用 new 处理函数的循环，但这个版本中则没有。但这些惯例是条目 51 的主题，所以此处暂时忽略它们。我要专注讨论一个更为细微的问题：**对齐**问题。

许多计算机架构需要使特定类型的数据放置在内存中特定类型的地址中。比如，某架构可能会要求指针的首地址为四的倍数（也就是**四字节对齐**）。不遵循这一惯例将会导致运行时发生硬件异常。其它架构可能更宽松些，但是如果做到数据对齐，它们的性能表现可能会更好。比如，double 在 Intel x86 架构中可在任意字节处对齐，但是如果将其按八字节对齐，那么访问它们的速度会大大加快。

此处之所以涉及对齐问题，是因为 C++要求所有 operator new 返回的指针对于任何类型数据都可对齐。malloc 亦致力于此，因此让 operator new 返回一个来自 malloc 的指针是安全的。但是，上文的 operator new 中，我们并没有返回来自 malloc 的指针，而是返回了在 malloc 返回值的基础上**偏移一个 int 尺寸**的指针。这样做的安全性是毫无保证的。如果客户调用 operator new 为 double（或者针对 double 数组的 operator new[]）分配了足够的内存，并且当前机器上的 int 占四个字节但是 double 需要八字节对齐，我们很可能会返回一个没有对齐的指针。这可能会导致程序崩溃。也可能仅仅导致运行速度降低。无论哪种结果都不是我们所希望的。

诸如内存对齐这类问题可以将内存管理专家和处理其他任务的程序员区分开。编写一个大致可运行的内存管理机制很简单，**完美的**版本则要复杂得多。因此我建议，一般情况下就不要去尝试了，除非有特殊需求。

大多数情况下都不存在特殊需求。一些编译器有预定的开关，可以打开内存管理函数的 debug 和记录功能。简要阅读一下编译器的文档便可消除编写 new 和 delete 的顾虑。在很多平台上，商业产品会替换编译器提供的内存管理函数。为了利用它们增强的功能，以及（可能带来的）性能的提升，你所需要的仅仅是重新连接程序。

（好吧，首先你得买下它们。）

另一个选择是开源的内存管理方案。它们支持多种平台，你都可以下载试用。Boost（参见条目 55）的内存池库就是一个开源的内存分配器。它提供了专为大量小尺寸对象调教的分配方案，这是自定义内存管理有价值的最普遍情形之一。许多 C++ 的书籍（包括本书早期版本）都提供了高性能小尺寸对象分配器的代码，但通常都会忽略诸如可移植性、内存对齐和线程安全等琐碎的细节。真实的库中代码应该更健壮。即使你准备自己编写 `new` 和 `delete`，对于一些区分“差不多可行”和“真正可行”的容易忽略的细节，阅读开源版本的分配器可能会让你了解更多。（内存对齐就是一个这样的细节，值得注意的是 TR1（参见条目 54）中添加了对查找特定类型对其需求的支持。）

本条目的主题是了解有必要取代默认 `new` 和 `delete` 的时机，无论是全局的还是基于某个类的。现在我们在更细节的层面来总结一下这些时机：

- **甄别使用错误。**（同上）
- **收集动态分配内存使用的统计数据。**（同上）
- **提高分配和释放的速度。**通用的分配器通常（并不绝对）要比自定义版本慢许多，若自定义版本是为某个特定类型设计的尤甚。特定类的分配器是固定尺寸分配器（比如 Boost 的内存池库中提供的）一次实例应用。如果你的程序是单线程的，但是编译器默认的内存管理规则是线程安全的，那么你编写线程不安全的分配器也可以赢得显著的速度提升。当然，认定程序中的 `operator new` 和 `operator delete` 值得进行性能改善之前，应先确认它们是否真的是瓶颈所在。
- **降低默认内存管理模式带来的空间开销。**通用内存管理机制通常（并不绝对）不仅仅比自定义版本速度慢，它们还会占用更多内存。这是因为它们经常会为每个分配区块都带来一些额外开销。为小对象调教的分配器（比如 Boost 的内存池库）可基本消除这些开销。
- **补偿默认分配器的局部最优对齐。**如前文所讲，在 x86 架构中访问八字节对齐的 `double` 是最快的。但是某些编译器提供的默认 `operator new` 对于动态

分配的 double 无法保证八位对齐。在这些情况下，使用一个可以保证八位对齐的 operator new 来取代默认版本可为程序带来显著的性能提升。

- **聚合相似对象。**如果你知道特定的数据结构一般都会协同使用，并且你期望在使用数据时页面错误出现的频率降至最低，那么为每种数据结构创建一个单独的堆就是明智的做法，这样可使相似的对象聚合在尽量少的分页中。new 和 delete 的布局化版本（参见条目 52）可以达成这样的聚合。

- **获取常规之外的新行为。**有时你需要让 operator new 和 delete 完成一些编译器版本无法完成的工作。比如，你可能需要在共享内存中分配和释放内存块，但是手头只有一套 C 的 API 可以管理共享内存。编写 new 和 delete 的自定义版本可以给这套 C 的 API 穿上一件 C++的外衣。再比如，你可以编写一个自定义版本的 operator delete 将释放的内存用零值覆盖，以提升程序数据的安全性。

时刻牢记

- 编写自定义版本的 new 和 delete 有许多合适的理由，包括提升性能、堆使用除错，以及收集堆使用信息，等等。

条目51： 在编写 new 和 delete 时要遵循常规

条目 50 中讨论了自行编写 operator new 和 operator delete 恰当的时机，但是并没有提及编写时应遵循哪些常规。遵守这些规则并不困难，但是其中一些实现起来略显琐碎，所以有必要了解它们。

我们从 operator new 开始。实现一个常规版本的 operator new 要有正确的返回值，在内存不足时要调用 new 处理函数（参见条目 49），随时准备应对零内存请求。你还应防止在不经意间覆盖了“正常”形式的 new，这更像一个类接口问题而不是一个实现需求，将在条目 52 中讲述。

operator new 的返回值部分很简单。如果你可以满足内存请求，你应返回一个指向它的指针，如果满足不了，你应该遵循条目 49 中描述的规则，抛出一个 bad_alloc 类型的异常。

然而事情也不是那么简单，因为 operator new 实际上会尝试多次分配内存，在每次失败后会调用 new 处理函数。此处假设 new 处理函数可以释放一部分内存。只有当指向 new 处理函数的指针为空时，operator new 才会抛出异常。

说起来奇怪，C++要求 operator new 总返回一个合法的指针，及时当前请求了零字节的内存。(这一古怪的行为可以简化 C++其它场合的某些问题）下面是基于此事实的一个非成员 operaor new 伪代码：

```
void* operator new(std::size_t size) throw(std::bad_alloc)
{                                    // 你的 operator new 可能有其他参数
  using namespace std;
  if (size == 0) {                   // 解决零字节请求问题：将其视为 1 字节请求
    size = 1;
  }
  while (true) {
    尝试分配 size 字节的内存;
    if (分配成功)
      return (指向所分配内存的指针);

    // 分配失败，找出当前 new 处理函数（参见下文）
    new_handler globalHandler = set_new_handler(0);
    set_new_handler(globalHandler);

    if (globalHandler) (*globalHandler)();
    else throw std::bad_alloc();
  }
}
```

把零字节请求看做 1 字节请求的戏法有些蹩脚，但是它简单、合法、可运行，并且你又能遇到多少次零字节请求呢？

你也会对伪代码的某部分感到怀疑，即把 new 处理函数的指针设为空，然后立即重置为原始状态。不幸的是，我们无法直接取得 new 处理函数的指针，因此你必须调用 set_new_handler 来指明该函数。简单粗暴而又不失高效性，至少在单线程代码中是这样的。在多线程环境中，你可能需要某种互斥锁来安全地处理 new 处理函数背后的（全局）数据结构。

条目 49 指出 operator new 中包含一个无限循环，上文的代码进行了显式展示。“while(true)”保证了循环的无限性。跳出循环的途径只有内存成功分配或 new 处理函数完成了条目 49 中描述的工作之一：使更多内存可用、安装另一个 new 处理函数、删除当前的 new 处理函数、抛出异常或集成 bad_alloc、或者出错返回。现在你清楚为什么 new 处理函数必须完成这些工作之一了。如果不去做，operator new 中的循环将永远不会终止。

许多人并没有意识到作为成员函数的 operator new 会被派生类所继承。这也就会导致一些有趣的编译结果。在上文中 operator new 的伪代码中，请注意那个尝试分配 size 字节（size 为零时除外）的函数。这样完全合理，因为 size 就是传入函数的参数。然而，正如条目 50 所讲，为特定类的对象（而不是某一个类，或者这个类的派生类）优化分配机制，是编写内存管理代码最普遍的理由之一。也就是说，为 X 类给定一个 operator new，一般情况下它将为 sizeof(X)尺寸的对象进行调教，不大不小。然而由于继承的存在，派生类对象中可能会调用基类的 operator new 来分配内存：

```
class Base {
public:
  static void* operator new(std::size_t size) throw(std::bad_alloc);
  ...
};

class Derived: public Base          // 派生类中没有声明 operator new
{ ... };

Derived *p = new Derived;           // 会调用基类的 operator new！
```

如果 Base 特定的 operator new 没有考虑到 Derived（很大程度上都不会），最好的解决方法就是阻止对“错误”尺寸内存的请求，转而交给标准 operator new 处理，代码如下：

```
void* Base::operator new(std::size_t size) throw(std::bad_alloc)
{
  if (size != sizeof(Base))         // 如果 size“错误”
    return ::operator new(size);    // 就由标准 operator new 处理该请求
```

```
  ...                                    // 否则，在这里处理请求。
}
```

“等等！”有朋友会说，“你忘了检查 size 为零这种病态但随时可能出现的情况了！”我的确没考虑，但是请你先喘口气。检查一直存在，只是在这里被收入了 size != sizeof(Base) 的麾下。C++总会剑走偏锋，认为所有独立对象的尺寸均不为零就是一个表现（参见条目 39）。sizeof(Base) 的尺寸从其定义层面讲绝不会为零，因此如果 size 为零，请求就会转向::operator new，它将担负起以合理方式处理请求的任务。

如果你想按“每个类”的形式来控制数组的内存分配，那么你需要实现 operator new 的数组特定的表亲——operator new[]。（这个函数通常被称为数组 new，因为“operator new[]”怎么读很难确定。）如果你想自己编写 operator new[]，请记住你要做的是分配一大块原始内存，你不能对数组中当前尚不存在的对象进行任何操作。事实上，你甚至无法了解数组中将会有多少个对象。首先，你不知道每个对象的大小。并且派生类对象可能通过继承来调用基类对象的 operator new[]，从而使得派生类对象一般都会比基类对象大。因此，你不能假定 Base::operator new[]出的数组中的每一个对象的尺寸都是 sizeof(Base)，这也就意味着你不能假定数组中对象的个数为“(请求的字节数)/sizeof(Base)”。其次，传递给 operator new[]的 size_t 参数的值可能会大于对象占用的内存，这是因为动态分配数组可能会包含存储数组元素数量的额外空间（参见条目 16）。

以上就是编写 operator new 需要遵循的所有常规了。对于 operator delete，情况更简单些。你要记住的只有一件事：C++确保释放空指针永远安全，因此你应该履行这项保证。以下是一个非成员 operator delete 的伪代码：

```
void operator delete(void *rawMemory) throw()
{
  if (rawMemory == 0) return;          // 如果当前删除的指针为空，就什么也不做
  释放 rawMemory 所指的内存;
}
```

成员版本也很简单。只是你需要确认所释放对象的尺寸。假设你的类特定版本的 operator new 将“错误”尺寸的请求传递给了::operator new，你应将“错误尺寸”的释放请求也传递给::operator delete：

```
class Base {
public:
  static void* operator new(std::size_t size) throw(std::bad_alloc);
  static void
    operator delete(void *rawMemory, std::size_t size) throw();
    ...
};

void Base::operator delete(void *rawMemory, std::size_t size) throw()
{
  if (rawMemory == 0) return;      // 空指针检测

  if (size != sizeof(Base)) {      // 如果 size“错误”,
    ::operator delete(rawMemory);  // 就由标准 operator delete 处理该请求
    return;
  }

  释放 rawMemory 所指的内存
  return;
}
```

有趣的是，如果当前正在释放的对象继承自一个没有虚析构函数基类，那么 C++ 传递给 operator delete 的 size_t 值则可能是错误的。这足以成为“基类必备虚析构函数”的理由了，但是条目 7 中描述了另一个更好的有依据的理由。现在只需注意：如果基类中忽略了虚析构函数，operator delete 函数可能会出错。

时刻牢记

- operator new 应包含一个无限循环以分配内存，应在其无法满足内存请求时调用 new 处理函数，同时应处理好零字节请求。类定制版本应能够处理大于预期的内存块请求。
- 如果传入一个空指针，则 operator delete 应什么也不做。类定制版本应能够处理大于预期的内存块。

条目52： 若用“布局 new”则必用“布局 delete”

布局 new 和布局 delete 并不是 C++潘多拉魔盒中那个最常见的邪术，因此如果你不了解它也不要担心。反之，请回忆条目 16 和 17，当你编写了下面的 new 语句：

```
Widget *pw = new Widget;
```

将调用两个函数：一个 operator new 来分配内存，另一个是 Widget 的默认构造函数。

假设两次调用都成功了，但是第二次调用中抛出了一个异常。此时 C++必须收回第一次调用中分配的内存。否则将导致内存泄漏。客户代码不能释放这样的内存，这是因为如果 Widget 的构造函数抛出异常，pw 将不会得到赋值。客户无法取得待释放内存的指针。回收第一次调用所分配内存的任务落在了 C++运行时系统的肩上。

运行时系统很乐意调用与第一步中调用的 operator new 相关的 operator delete。但是它只在知晓哪一个（可能有很多）operator delete 是恰当版本时才会那样做。如果当前的 new 和 delete 使用正常的签名，那么这并不是一个问题。因为正常的 operator new 和正常的 operator delete 也是相关联的：

```
void* operator new(std::size_t) throw(std::bad_alloc);
                                    // 全局的正常签名
void operator delete(void *rawMemory) throw();
                                    // 全局的正常签名
void operator delete(void *rawMemory, std::size_t size) throw();
                                    // 典型的、class 域的正常签名
```

因此，如果你只使用 new 和 delete 的正常形式，运行时系统就可以找到那个知晓如何回收 new 出内存的 delete。但是如果你声明了非正常形式的 operator new（即有更多参数的形式），“此 new 适用哪个 delete”的问题就出现了。

举例说，假设你编写了一个特定类的 operator new，可为其指明一个 ostream 来记录分配信息，你也需要编写一个正常的特定类的 operator delete：

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size, std::ostream& logStream)
    throw(std::bad_alloc);          // new 的非正常形式

  static void operator delete(void *pMemory, std::size_t size)
    throw();                        // delete 的正常的类定制形式
  ...
};
```

这一设计会带来问题，但是在寻找原因之前，需要插播一小段术语简介。

当 operator new 函数有额外的参数（除强制的 size_t 外的参数）时，人们将其称为“布局 new”。上文代码中的 operator new 就是一个布局 new。有一种布局 new 非常有用，它们取一个指针参数来指明对象构造的方位。这种 operator new 大致是这样的：

```
void* operator new(std::size_t, void *pMemory) throw();     // “布局 new”
```

这一版的 new 来自 C++标准库，可以通过#include<new>随时使用。这版的 new 在 vector 内部使用，可在 vector 未使用的空间上创建对象。它还出现于各种场合。也是最原始的布局 new。事实上，这也是这个函数出名的原因，即：**作为布局 new**。这也就意味着“布局 new”这个术语可以重载。多数时候人们讨论布局 new 时,他就讨论的就是这个特定的函数，即包含一个多余的 void*类型参数的 operator new。上下文基本上可以消除所有歧义，但是“布局 new”这个一般术语意味着所有包含额外参数的 new 版本，理解这一点很重要，因为“布局 delete”的说法直接由它继承而来，我们稍后再讲。

但是让我们先回到 Widget 的定义，就是我说会引起问题的那个。难点在于这个类会导致难以察觉的内存泄漏。请考虑以下的客户代码，其中在动态创建一个 Widget 时通过 cerr 记录了分配信息：

```
Widget *pw = new (std::cerr) Widget; // 调用 operator new,
                                     // 将 cerr 作为 ostream 传递,
                                     // 若 Widget 的构造函数抛异常，将泄漏内存
```

再次强调，如果内存分配成功，且 Widget 的构造函数抛出一个异常，运行时系统将负责回收 operator new 分配的内存。然而，运行时系统并不能够真正理解所调用的 operator new 是怎样工作的，因此它无法自行完成回收工作。运行时系统将转而寻找一个与 operator new 拥有同样数量和类型的额外参数的 operator delete，如果找到了，就调用它。在此情况下，operator new 包括一个 ostream&类型的额外参数，因此相关的 operator delete 将拥有下面的签名：

```
void operator delete(void*, std::ostream&) throw();
```

我们了解了布局 new，依此类推，包括额外参数的 operator delete 版本可称为布局 delete。此情况中，Widget 并没有声明任何布局 delete，因此运行时系统就无从得知如何撤销布局 new 的调用。于是它什么也不会做。在此示例中，如果 Widget 的构造函数抛出异常，则**不会调用任何 operator delete**！

规则十分简单：如果包含额外参数的 operator new 与一个包含同样额外参数的 operator delete 之间不是成对出现的，那么在 new 出的内存需要撤销时就没有可供调用的 operator delete。为了避免上面代码中的内存泄漏，Widget 应该声明一个与那个做记录用的布局 new 相匹配的布局 delete：

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size, std::ostream& logStream)
    throw(std::bad_alloc);
  static void operator delete(void *pMemory) throw();
  static void operator delete(void *pMemory, std::ostream& logStream)
    throw();
  ...
};
```

有了这一改变，即使在下面的语句中：

```
Widget *pw = new (std::cerr) Widget;     // 同上，但这里不会泄漏内存
```

Widget 的构造函数抛出了异常，相应的布局 delete 也会得到自动调用，这使得 Widget 可以确保不泄漏内存。

然而，请考虑如果程序没有抛出异常，而客户代码遇到这样的 delete 时，将会发生什么：

```
delete pw;                          // 调用正常的 operator delete
```

就像注释所说，这里会调用正常的 operator delete，而不是布局版本。只在与布局 new 调用相关的构造函数调用发生异常时，才会调用布局 delete。delete 一个指针（如上文的 pw）**永远**也不会带来布局 delete 的调用。

这也就意味着：为了避免布局 new 相关的内存泄漏，你必须同时提供正常的 operator delete（构造过程中不抛异常时使用）和一个包含与 operator new 有相同额外参数的布局 delete（抛异常时使用）。这样，对隐蔽的内存泄漏问题，你终于可以高枕无忧了。嗯，至少这一类是可以放心了。

顺便讲一下，由于成员函数的名字会覆盖外围作用域的同名函数（参见条目 33），你需要小心避免让特定类的 new 覆盖客户期望的其它 new（包括正常版本）。比如，如果你让一个基类只声明一个布局 new，那么客户将无法使用 new 的正常版本：

```
class Base {
public:
  ...
  static void* operator new(std::size_t size, std::ostream& logStream)
     throw(std::bad_alloc);         // 这个 new 将覆盖正常的全局版本
  ...
};

Base *pb = new Base;                // 错误！new 的正常版本已被隐藏
Base *pb = new (std::cerr) Base;    // 正确，调用 Base 的布局 new
```

类似地，派生类中的 operator new 会将全局的和派生版本的 operator new 都隐藏起来：

```
class Derived: public Base {        // 继承自上文的 Base 类
public:
  ...
  static void* operator new(std::size_t size)
    throw(std::bad_alloc);          // 重新声明正常形式的 new
```

```
  ...
};

Derived *pd = new (std::clog) Derived;
                                    // 错误！Base 的布局 new 被隐藏了
Derived *pd = new Derived;          // 正确，调用 Derived 的 operator new
```

条目 33 中非常详细地讨论了这类名字隐藏机制的细节，但是为了编写内存分配函数，你需要记住：C++在默认情况下会提供以下三种全局 operator new：

```
void* operator new(std::size_t) throw(std::bad_alloc);
                                    // 正常的 new
void* operator new(std::size_t, void*) throw();
                                    // 布局 new
void* operator new(std::size_t, const std::nothrow_t&) throw();
                                    // 不抛异常的 new（参见条目 49）
```

如果你在一个类中声明了任意的 operator new，就会覆盖上述所有的标准形式。除非你有意阻止类的客户去使用这些形式，你应该在所有自定义的 opeator new 形式以外保证这些形式都可用。当然，对于你提供的每个 operator new，都应该提供相应的 operator delete。如果你期望这些函数能够正常运行，只需让类定制版本去调用全局版本就行了。

完成此工作的一个简便方法是：创建一个包含所有 new 和 delete 正常形式的基类：

```
class StandardNewDeleteForms {
public:
  // 正常的 new/delete
  static void* operator new(std::size_t size) throw(std::bad_alloc)
  { return ::operator new(size); }

  static void operator delete(void *pMemory) throw()
  { ::operator delete(pMemory); }

  // 布局 new/delete
  static void* operator new(std::size_t size, void *ptr) throw()
```

```
  { return ::operator new(size, ptr); }

  static void operator delete(void *pMemory, void *ptr) throw()
  { return ::operator delete(pMemory, ptr); }

  // 不抛异常的 new/delete
  static void* operator new(std::size_t size, const std::nothrow_t& nt)
    throw()
  { return ::operator new(size, nt); }
  static void operator delete(void *pMemory, const std::nothrow_t&)
    throw()
  { ::operator delete(pMemory); }
};
```

对于想为标准形式中添加自定义形式的客户，现在就可以通过继承和 `using` 声明来取得标准形式：

```
class Widget: public StandardNewDeleteForms {          // 继承自标准形式
public:
  using StandardNewDeleteForms::operator new;           // 使这些标准形式
  using StandardNewDeleteForms::operator delete;        // 可见

  static void* operator new(std::size_t size,
                            std::ostream& logStream)    // 添加自定义的
    throw(std::bad_alloc);                              // 布局 new

  static void operator delete(void *pMemory,
                              std::ostream&logStream)   // 添加相应的
    throw();                                            // 布局 delete
  ...
};
```

时刻牢记

- 在编写布局 `new` 时，应编写相应的布局 `delete`。否则你的程序会出现难以察觉的、断断续续的内存泄漏。

- 在声明一对布局 `new` 和 `delete` 时，不要故意覆盖它们的正常版本。

第九章. 杂集

欢迎阅读“杂集”，本章的主题是查漏补缺。虽然只有三个条目，但是不要因为条目少或者标题没逼格就忽略它们。本章十分重要。

第一个条目中强调了：只要你想让软件正确运行，那么编译器的警告信息便忽视不得。第二个条目是对 C++标准库的概述，其中包含了 TR1 中引入的重要新功能。最后一个条目是对 Boost 的概述，boost.org 是公认的最重要的全能 C++网站。如果缺少这三个条目的信息，编写高效 C++软件将会事倍功半。

条目53： 留心编译器的警告信息

许多程序员都会日常忽略编译器警告。毕竟“如果问题严重，我将得到一条出错信息”，不是吗？这样的想法在其它语言中可能相对正确，但 C++这里，我敢打赌编译器的作者可比你明白多了。比如，以下是一个大家在不经意间都会犯下的错误：

```
class B {
public:
  virtual void f() const;
};

class D: public B {
public:
  virtual void f();
};
```

你的思路是，让 `D::f` 重定义虚函数 `B::f`，但是此处有一个错误：`B` 中的 `f` 是一个 `const` 成员函数，但是 `D` 中没有声明为 `const`。我知道的一款编译器对此会给出以下警告：

```
warning: D::f() hides virtual B::f()
```

初级程序员对此信息大都会自言自语：“我还不知道 `D::f` 会覆盖 `B::f`？这本来就是我要干的活啊。”你错了。编译器是想告诉你：`B` 中声明的 `f` 并没有在 `D` 中得到重新声明，而是被完全覆盖了（条目 33 解释了这一现象）。忽略这一编译器警告几

乎必定会给程序带来错误的行为，以及随之而来的大量的调试工作，只为了找到那个编译器一开始就发现的问题。

在你对某款特定的编译器积累了丰富的经验后，你会理解不同信息的含义（往往与**字面**意思相去甚远）。一旦有了足够的经验，你可以选择忽略一系列警告信息，尽管一般认为编写无警告的代码是更好的，即使是最高级别的警告。无论如何，你很有必要在忽略警告之前先弄清它真正要告诉你的事情。

鉴于我们的主题是警告信息，请记住警告信息是平台独立的，这是由编译器内在实现方式决定的，因此随随便便编写程序，然后把除错工作都甩给编译器并不是一个好主意。比如，上文中覆盖函数的代码在另一个编译环境中会毫无抱怨的顺利通过。

时刻牢记

- 认真对待编译器的警告信息，在你的编译环境的最高警告级别下力争做到零警告。
- 别完全依赖编译器警告信息，因为不同的编译器作出警告的角度是不同的。如果你依赖某些警告，而你转向新的编译器时可能不存在此类警告。

条目54： 熟悉标准库，包括 TR1

C++标准（定义语言本身和库的文档）批准于 1998 年。2003 年发行了一个小型的“错误修订”更新。然而标准委员会并没有停滞不前，预计“2.0 版”C++标准将于 2009 年面世（尽管所有实际工作预计将在 2007 年完成）。直到最近，下一版 C++面世的年份仍未确定，这也就解释了为什么人们通常称其为“C++0x”，即 200x 年版 C++。

C++0x 可能会包含一些有趣的全新语言特征。但是大多数新功能将以扩展标准库的形式实现。我们已经了解了新版标准库中一些新功能，因为这些功能已经在 TR1（“Technical Report 1”,C++库工作组的“技术报告 1”）文档中进行了描述。标准委员会保留了在 C++0x 官方收录 TR1 前对其进行修改的权利，但是不会有大的改动了。简言之，TR1 将会为 C++带来一个全新的版本，我们可以称之为 C++ 1.1。TR1 的功能惠及类和应用的方方面面，如果你不去了解它，那么你就不能成为一个高效 C++程序员。

在学习 TR1 之前，有必要先复习一下 98 版 C++标准库的主要部分：

- **标准模板库（STL）**，其中包括容器（vector、string、map 等）、迭代器、算法（find、sort、transform 等）、函数对象（less、greater 等），以及多种容器核函数对象的变种（stack、priority_queue、mem_fun、not1 等）。
- **输入输出流（iostream）**，包括对用户定义缓冲区、国际化 IO、以及预先定义好的对象（cin、cout、cerr、clog）的支持。
- **国际化支持（I18N）**，以及多区域支持的能力。诸如 wchar_t 和 wstring 的类型使 Unicode 编码更方面。
- **数值处理支持**，包括复数模板（complex）和纯数值数组（valarray）。
- **异常处理层次结构**，包括基类 exception 及其派生类 logic_error 和 runtime_error，以及后二者的各种派生类。
- **C89 标准库**。1989 版 C 标准库中的一切都在 C++中。

如果你对上文中任一部分感到陌生，那么我建议你安排一些高质量的时间，阅读一下你最喜欢的 C++参考书，以改善一下现状。

TR1 由 14 部分组成（也就是说库的功能分为 14 大块）。都收录在 std 名字空间中，更精确的说是收录在 std 内嵌的 tr1 名字空间中。于是 TR1 组件 shared_ptr（参见下文）的全名就是 std::tr1::shared_ptr。本书中，在讨论标准库的组件时我会习惯性省略 std::，但是始终保留 TR1 组件的前缀 tr1::。

本书中的实例涉及了以下 TR1 的组件：

- **智能指针**，tr1::shared_ptr 和 tr1::weak_ptr。tr1::shared_ptr 拥有内建指针的行为，但是它们会跟踪有多少 tr1::shared_ptr 指向某个对象。这一特性被称为“**引用计数**”。在最后一个这样的指针被销毁后（也就是说在某个对象的引用计数值为零时），对象被自动释放。这可以有效防止非循环数据结构的资源泄漏，但是如果两个或更多包含 tr1::shared_ptr 的对象组成了一个循环，那么这个循环将使得其中所有的对象的引用计数都无法归零，即使所有指向这个循环的外部指针都被销毁了（即这一组对象无法访问时）。

于是 tr1::weak_ptr 登场了。tr1::weak_ptr 可在 tr1::shared_ptr 型数据结构有问题时作为循环引导指针。tr1::weak_ptr 不参与引用计数。当指向某一对象的最后一个 tr1::shared_ptr 销毁后，无论是否还存在 tr1::weak_ptr，对象都将释放。然而这样的 tr1::weak_ptr 将自动被标记为无效。

tr1::shared_ptr 可能是 TR1 中用途最广泛的组件。本书中我曾多次使用它，在条目 13 中，我曾解释了它的重要性。(本书中没有使用 tr1::weak_ptr，抱歉。)

- **tr1::function。**可表示所有原始签名与目标签名保持不变的**可调用实体**（即函数或函数对象）。如果我们期望注册一个以 int 为参数、返回一个 string 的回调函数，可以这样做：

```
void registerCallback(std::string func(int));
                                  // 参数类型是一个
                                  // 以 int 为参数、返回一个 string 的函数
```

参数名 func 可省略，因此 registerCallback 也可以这样声明：

```
void registerCallback(std::string (int));
                                  // 同上，省略参数名
```

请注意此处的“std::string (int)”是一个函数签名。

tr1::function 使 registerCallback 可接受所有以 int（或者**所有可由 int 转换出的类型**）为参数、返回一个 string（或者**所有可转换为 string 的对象**）的可调用实体，从而使其变得更灵活。tr1::function 以其目标函数的签名作为模板参数：

```
void registerCallback(std::tr1::function<std::string (int)> func);
                                  // 所有签名恒为“std::string(int)”的
                                  // 可调用实体，参数 func 都能接受
```

这一灵活性及其有用，我已在条目 35 中详细展示。

- **tr1::bind。**除了可以完成 STL 中绑定器 bind1st 和 bind2nd 的所用工作外，还能做更多。与前 TR1 绑定器不同，tr1::bind 可同时应用于 const 与

非 const 的成员函数。它还可以在无需帮助的情况下处理函数指针，从而使得在调用tr1::bind前也无需调用ptr_fun、mem_fun或者mem_fun_ref等等函数。简言之，tr1::bind是全新一代绑定器，它比前一代有了长足进步。条目 35 中有一个相关的实例。

我将剩余的 TR1 组件分为两大部分。第一组提供彼此独立的功能：

- **哈希表。**用于实现 set（按照键排序的唯一键集合）、multiset（按照键排序的键集合）、map（按照键排序的键值对集合，键唯一）、multimap（按照键排序的键值对集合）。每个新容器都拥有一套与之前版本相对应的接口。最让人意想不到的事 TR1 中哈希表的命名：tr1::unordered_set、tr1::unordered_multiset、tr1::unordered_map 以及 tr1::unordered_multimap。这些名字强调了，与 set、multiset、map 或 multimap 不同，TR1 中的哈希容器没有按任何可预见的方式排序。

- **正则表达式。**引入了基于正则表达式的字符串查找替换操作，通过匹配操作遍历字符串，等等能力。

- **多元组（tuple）。**标准库已经对 pair 模板做出了一项实用的更新。但是 pair 对象只能保存两个对象，而 tr1::tuple 可以保存任意数量的对象。逃离 Python 和 Eiffel 的程序员们，你们又能找回自信了！你们之前家园现在也是 C++ 的一部分。

- **tr1::array。**大致上就是一个“STL 化”的数组，也就是一个支持诸如 begin 和 end 这样的成员函数的数组。tr1::array 的尺寸在编译时是固定的，对象不涉及动态内存。

- **tr1::mem_fn。**在语法上与成员函数指针一致。二者就好比 tr1::bind 对 C++98 的 bind1st 和 bind2nd 功能的归入和扩展。tr1::mem_fn 也是归入并扩展了 C++98 的 mem_fun 和 mem_fun_ref 的功能。

- **tr1::reference_wrapper。**使引用更像对象的一个功能。它的功能之一是可以创建（似乎能够）存储引用的容器。（事实上，容器只能存储对象或指针。）

- **随机数生成器。**比从 C 标准库中继承而来的 rand 函数有显著的提升。

- **数学专用函数。**包括拉盖尔多项式、贝塞尔函数、完全椭圆积分，等等等等。
- **C99 兼容扩展。**一组用于为 C++提供 C99 库中丰富的新功能的函数和模板。

TR1 组件的第二部分包含了更多复杂而精确的模板编程技术的支持，其中包括模板元编程（参见条目 48）：

- **类型特征（type traits）。**一系列特征类（参见条目 47），可提供类型的编译时信息。给定一个类型 `T`，TR1 的类型特征功能可以指明 `T` 是否为内建类型、是否提供虚析构函数、是否是一个空类（参见条目 39）、是否可以隐式转换为其他的类型 `U`，等等。TR1 的类型特征功能也可以为一个类型指明恰当的对齐信息，这对于需要编写自定义内存分配函数的程序员来说至关重要（参见条目 50）。
- **`tr1::result_of`。**可推断出函数调用返回值的模板。在编写模板时，经常需要取得某个函数（或模板）返回值的类型，但是返回值依赖于函数参数的类型，关系十分复杂。`tr1::result_of` 简化了取得函数返回值类型的方法。TR1 库自身也存在许多对 `tr1::result_of` 的应用。

尽管 TR1 中某些功能已经可以取代早期的组件（尤其是 `tr1::bind` 和 `tr1::mem_fn`），它仍是对标准库一次纯粹的更新。其中的任何组件也没有取代现有的组件，因此由前 TR1 组件编写的上古代码依然有效。

TR1 只是一个文档。为了使用它所提供的功能，你需要访问它的实现代码。这些代码最终将会与新版编译器捆绑发布，但是在我于 2005 年写下这段话的时刻，如果你在你的标准库实现中寻找 TR1 组件，很可能会有缺失。幸运的是，你还可以到其它地方去寻找：14 个组件中的 10 个是基于 Boost 库（参见条目 55）的，你可免费下载，所以我们说 Boost 是一个极好的“类 TR1”功能资源。我说“类 TR1”，是因为尽管许多 TR1 功能都是基于 Boost 库的，但是当前还有很多 Boost 功能与 TR1 说明文档不完全匹配的地方。很有可能在你阅读这一段时，Boost 不仅会对 TR1 中由 Boost 进化而来的组件提供与 TR1 一致的实现，同时也会提供 TR1 中原本不是基于 Boost 的四个 TR1 库的支持。

如果你想在编译器提供内建的 TR1 支持前，为使用 Boost 的“类 TR1”提供权宜之计，那么你需要使用一个名字空间的小戏法了。所有 Boost 组件都在 boost 名字空间

内，但是 TR1 组件都在 std::tr1 中，你可以告诉编译器一致对待 std::tr1 和 boost。以下是方法：

```
namespace std {
  namespace tr1 = ::boost;          // std::tr1 名字空间是 boost 名字空间的别名
}
```

从技术上讲，这使得你陷入未定义行为的危险，因为为 std 名字空间添加任何都系都是不允许的。在实践中，你并不会遇到麻烦。当你的编译器提供它们自己的 TR1 实现时，你所需要做的只是删除上文的名字空间别名的设定；引用 std::tr1 的代码将依旧合法。

TR1 最重要的不基于 Boost 的部分可能就是哈希表，但是哈希表已经以 hash_set、hash_multiset、hash_map 和 hash_multimap 的名字在多种代码中存在了很多年了。你的编译器内建的库中很可能包含这些模板。如果找不到，发动你最常用的搜索引擎来查找一下这些名字吧（还有 TR1 版本的名字），因为无论在商业还是自由软件中，你一定会找到包含它们的代码。

时刻牢记

- C++标准库主要包括 STL、输入输出流、地区设定，以及 C89 标准库。
- TR1 增加了智能指针（比如 tr1::shared_ptr）、一般化函数指针（tr1::function）哈希表容器、正则表达式，以及其它十项组件的支持。
- TR1 自身只是一个说明文档。为了利用它的功能，你需要一个实现版本。TR1 组件的一个实现来源是 Boost。

条目55： 熟悉 Boost

想找一组高质量、开源、跨平台、跨编译器的库？请看 Boost。有兴趣加入一个由一群有理想、有天赋 C++程序员组成的社区，与他们一同设计和实现艺术水准的类库吗？请看 Boost。想领略未来 C++的风采吗？依然请看 Boost。

Boost 是一个 C++开发者社区，也是一组可免费下载的 C++类库。网址是 http://boost.org。快把它放进收藏夹吧。

固然，C++有众多组织和网站，但是 Boost 具有两项其他组织无可比拟的优势。第一，它与 C++标准委员会之间有着紧密和富有影响力的关系。Boost 就是由委员会成员成立的，二者的成员至今仍有很大重叠。此外，Boost 设立的目标之一，就是为即将加入标准 C++的功能提供测试环境，这一目标始终未变。于是这一关系带来了一个效应：即将通过 TR1 加入 C++的 14 个新类库中，多于三分之二都基于 Boost 的工作成果。

Boost 的第二个特殊优势是它添加新库的程序。Boost 基于公开的同行评审。如果你想为 Boost 添加一个库，你应首先在 Boost 开发者邮件列表上发布一条信息，以评估人们对这个类的关注度，并对作品进行前期审查。然后就开始了这个网站所谓“讨论、改进、重新提交，直至达标”的循环过程。

最终，由你自己决定你的类库是否符合提交的要求。一个评委将确定你的类库是否符合 Boost 的最低要求。比如，它必须在两款编译器下编译通过（以示正常的可移植性），并且你必须证明类库可以在一个可接受的授权下运行（比如，类库必须允许免费商业用途和非商业用途）。然后你提交的作品会出现在 Boost 社区供官方审核。在审核期内，志愿者会详细研究这个库的内容（包括源代码、设计文档、用户文档，等等）并提出类似下面的问题：

- 设计和实现足够好吗？
- 代码在不同的编译环境和操作系统下的可移植性如何？
- 目标用户（也就是类库所针对的工作领域的人们）会对这个类库感兴趣吗？
- 文档是否清晰、完整，并且精确？

这些评论都将在 Boost 邮件列表中发布，于是审核者和其他人都可以看到并且对其他人评价作出回应。在审核期结束后，将由评委决定你的类库是通过、部分通过，还是未通过。

同行评审很好地为 Boost 过滤掉了糟糕的类库，同时也教育了类库作者，符合工业标准的、跨平台的类库必定会在设计、实现和文档这些方面下足功夫。许多类库在通过前都经历了不止一次的官方评审。

Boost 包含许多库，而且在不断扩充中。偶尔也会移除一些库，一般都是因为它们

的功能已被一个功能更强大或设计更优秀（比如更轻便或更高效）的新库所取代。

库的尺寸和作用域分布广泛。一个极致的例子是某个库在概念上只需要几行代码（但是一般在添加了错误处理和可移植性支持后又变得很大了）。其中之一就是**类型转换库**（**Conversion**），它提供了更安全或更方便的转型运算符。比如它的numeric_cast 函数，如果在对一个数值进行类型转换时发生了溢出或反向溢出或者类似问题，这个函数将抛出一个异常。lexical_cast 使得所有支持 operator<<的类型都能够转换为字符串，这对于诊断和记录等非常有用。另一个极致的例子是某个库提供了非常广泛的功能，一个库够写一本书的。其中包括 **Boost 图库**（**Boost Graph Lbirary**，为各种图结构程序而设计）和 **Boost MPL 库**（“元编程库”，“metaprogramming library”）。

Boost 多样的类库仿佛一场饕餮盛宴，可以大致分为若干种类。其中包括：

- **字符串和文字处理。**包括用于类型安全的类 printf 格式化库、正则表达式（TR1 中类似功能的基础，参见条目 54）、字符串标记和语法分析。
- **容器。**包括采用 STL 接口的定长数组（参见条目 54）、变长的位集（bitset）和多维数组。
- **函数对象和高阶编程。**包括一些作为 TR1 功能的基础的库。一个有趣的库是兰姆达（Lambda）库，用它我们可以在不知不觉中随时随地轻松创建函数对象：

```
using namespace boost::lambda;      // 引入 boost::lambda 功能

std::vector<int> v;
...
std::for_each(v.begin(), v.end(), std::cout << _1 * 2 + 10 << "\n");
                                    // 对于 v 中的每个对象 x 打印 x*2+10，
                                    // “_1”是兰姆达库中当前元素的占位符
```

- **泛型编程。**包括大量的特征类。（对于 traits 的信息请参见条目 47。）
- **模板元编程**（TMP，参见条目 48）。包括一个编译时断言的库，和 Boost MPL 库。MPL 的用途之一就是：对**类型**等编译时实体提供“类 STL”的数据结构支持。

```
// 创建一个类似 list 的编译时容器，
```

```
// 包含三个类型（float、double 和 long double），称其为“floats”、
typedef boost::mpl::list<float, double, long double> floats;

// 创建一个新的编译时类型 list，
// 将“floats”和“int”一并添加进表头，称新容器为“types”
typedef boost::mpl::push_front<floats, int>::type types;
```

此类类型容器（通常称为**类型列表**，尽管它们除了可以基于 `mpl::list` 实现也可以基于 `mpl::vector`。）使 TMP 程序的世界豁然开朗。

- **数学和数字。**包括有理数、八元数和四元数、最大公约数和最小公倍数、以及随机数（又一个影响了 TR1 中相关功能的）库。
- **正确性和测试。**包括用于确定隐式模板接口（参见条目 41）库和用于改进测试程序的库。
- **数据结构。**包括类型安全的联合体库和（引领了 TR1 相关功能的）多元体库。
- **跨语言支持。**包括一个可以让 C++和 Python 无缝协作的库。
- **内存。**包括用于高性能定长分配器的内存池库（参见条目 50）、多种智能指针（参见条目 13），其中包括（但不限于）TR1 中的智能指针。scoped_array 就是一个非 TR1 的智能指针，它与 auto_ptr 类似，但应用于动态分配数组，条目 44 中一个示例使用过。
- **其它。**包括 CRC 校验库、日期和时间操作库以及文件系统遍历库。

记住以上列表只是 Boost 的沧海一粟。并不是全面的清单。

Boost 是一个多面手，但它也没有做到面面俱到。比如它没有 GUI 开发库，没有用于访问数据库的库。至少现在（我写下这段话时）没有。然而在你读到这里是可能已经有了。要想知道梨子的滋味就要亲口尝一尝。我建议你现在就行动起来：http://boost.org。即使你没有找到最初想找到的东西，Boost 总会给你惊喜。

时刻牢记

- Boost 是一个社区，一个网站，致力于自由、开源、经同行评审的 C++类库开发。Boost 在 C++标准化工作中占据重要地位。
- Boost 为许多 TR1 组件提供了实现，同时还提供更多其它类库。

附录 A. 拓展阅读

本书包括了我认为对于 C++从业人员最重要的基本守则，但是如果你希望进一步提高工作效率，我建议你看看我的另两本书：《More Effective C++》和《Effective STL》。

《More Effective C++》包含了额外的编程准则，并且对效率和异常编程等主题进行了更完整的讲解。同时也描述了诸如智能指针、引用计数和代理对象等 C++编程技术。

《Effective STL》与《Effective C++》一样，是一本编程守则导向的书籍。但它专注于如何高效实用标准模板类。

以下是这两本书的目录。

More Effective C++目录

基础

条目1： 区分指针和引用
条目2： 多用 C++风格的转型
条目3： 不要把数组多态化
条目4： 避免不必要的默认构造函数

操作符

条目5： 警惕用户自定义的类型转换函数
条目6： 区分自加自减运算符的前置形式和后置形式
条目7： 永远不要重载`&&`、`||`或`,`
条目8： 理解 `new` 和 `delete` 的不同含义

异常

条目9： 使用析构函数避免资源泄漏
条目10： 避免构造函数发生资源泄露
条目11： 避免在离开析构函数时发生异常

效率

技术

杂集

Effective STL 目录

附录 B. 第二版与第三版的条目映射表

本书第三版与第二版有很多不同，最明显的就是第三版中包含了许多新信息。然而，第二版的大部分内容都保留在了第三版中，只是改变了形式和所在位置。下文的两个表格中展示了第二版和第三版的信息是如何相互映射的。

表格展示的是**信息**的对应，而不是文字。比如，第二版中条目 39（“避免继承结构的向下转型”）位于当前版本的条目 27 中（“降低转型次数”），第三本中的文字和示例是全新的。一个更极致的示例涉及到第二版的条目 18（“类的接口要力争做到小而全”）。这一条目的首要结论之一就是：潜在的不需要对类的非公有部分进行特定访问的成员函数，一般都应是非成员。在第三版中，我们用了不同的（但更强有力的）推理方式得出了同样的结论，因此，即使两个条目之间的唯一相同之处就是它们的结论，我们依然将第二版的条目 18 映射至第三版的条目 23（“多用非成员非友元函数，少用成员函数”）。

第二版之于第三版

第二版	第三版	第二版	第三版	第二版	第三版
1	2	18	23	35	32
2	—	19	24	36	34
3	—	20	22	37	36
4	—	21	3	38	37
5	16	22	20	39	27
6	13	23	21	40	38
7	49	24	—	41	41
8	51	25	—	42	39, 44
9	52	26	—	43	40
10	50	27	6	44	—
11	14	28	—	45	5
12	4	29	28	46	18
13	4	30	28	47	4
14	7	31	21	48	53
15	10	32	26	49	54
16	12	33	30	50	—
17	11	34	31		

第三版之于第二版

第三版	第二版	第三版	第二版	第三版	第二版
1	—	20	22	39	42
2	1	21	23, 31	40	43
3	21	22	20	41	41
4	12, 13, 47	23	18	42	—
5	45	24	19	43	—
6	27	25	—	44	42
7	14	26	32	45	—
8	—	27	39	46	—
9	—	28	29, 30	47	—
10	15	29	—	48	—
11	17	30	33	49	7
12	16	31	34	50	10
13	6	32	35	51	8
14	11	33	9	52	9
15	—	34	36	53	48
16	5	35	—	54	49
17	—	36	37	55	—
18	46	37	38		
19	pp. 77–79	38	40		