



学习在线

视频资料下载
电子图书交流

www.eimhe.com

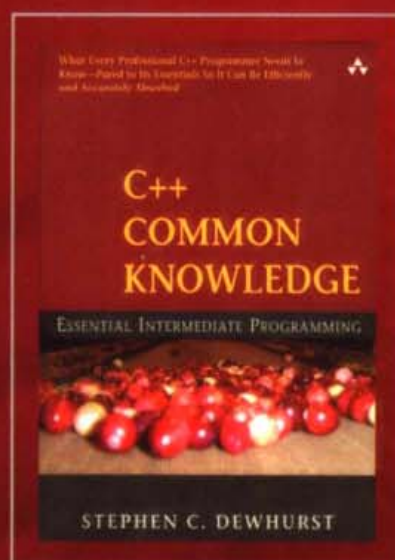
C++ Common Knowledge

Essential Intermediate Programming

C++ 必知必会

[美] Stephen C. Dewhurst 著
荣耀 译

- 职业 C++ 程序员必备常识
- 初学者登堂入室的阶梯
- C++ 界 20 年经验结晶



人民邮电出版社
POSTS & TELECOM PRESS

TURING

图灵程序设计丛书

C++ 必知必会

C++ Common Knowledge

Essential Intermediate Programming

[美] Stephen C. Dewhurst 著

荣 耀 译



人民邮电出版社

POSTS & TELECOM PRESS

图书在版编目 (CIP) 数据

C++必知必会 / (美) 杜赫斯特 (Dewhurst, S.C.) 著; 荣耀译.

—北京: 人民邮电出版社, 2006.1

(图灵程序设计丛书)

ISBN 7-115-14101-0

I. C... II. ①杜...②荣... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2005) 第 120478 号

内 容 提 要

本书描述了 C++ 编程和设计中必须掌握但通常被误解的主题, 这些主题涉及的范围较广, 包括指针操作、模板、泛型编程、异常处理、内存分配、设计模式等。作者根据本人以及其他有经验的管理人员和培训老师的经验总结, 对与这些主题相关的知识进行了精心挑选, 最终浓缩成 63 条。每一条款所包含的内容均为进行产品级 C++ 编程所需的关键知识。作者称这些知识为 C++ 程序员必备的“常识”, 其实并非意味着简单或平庸, 而是“必不可少”。

本书适合于中、高级 C++ 程序员, 也适合 C 或 Java 程序员转向 C++ 程序设计时参考。

图灵程序设计丛书

C++必知必会

-
- ◆ 著 [美] Stephen C. Dewhurst
 - 译 荣 耀
 - 责任编辑 陈贤舜
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子函件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京顺义振华印刷厂印刷
 - 新华书店总店北京发行所经销
 - ◆ 开本: 800×1000 1/16
 - 印张: 13.75
 - 字数: 289 千字 2006 年 1 月第 1 版
 - 印数: 1—5 000 册 2006 年 1 月北京第 1 次印刷

著作权合同登记号 图字: 01-2005-3575 号

ISBN 7-115-14101-0/TP · 5034

定价: 29.00 元

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

版权声明

Authorized translation from the English language edition, entitled: *C++ COMMON KNOWLEDGE: ESSENTIAL INTERMEDIATE PROGRAMMING*, 1st Edition, 0321321928 by DEWHURST, STEPHEN C., published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2005 Pearson Education, Inc.

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

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and Posts & Telecommunications Press Copyright © 2005.

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

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

对本书的赞誉

“我们正赶上 C++ 佳作不断涌现的时代，本书不愧为其中的一员。尽管 C++ 屹立于软件创新和生产力的最前沿已有 20 余年，但时至今日它才被充分理解和运用。这是一本难得的值得 C++ 实践者和专家反复研习的佳作。它并非那种晦涩难懂的学术气的论文，而是着力帮助你完善对 C++ 精髓的理解——这些都是你自以为已知但只有真正掌握否则早晚会栽跟头的重要知识。很少有人能像 Steve 这样精通 C++ 和软件设计。在软件开发方面几乎无人具备 Steve 这般冷静的头脑。他知道你需要学习了解些什么。相信我！当他发话时，我总是洗耳恭听。我希望你也能这样。你（和你的客户）终将为此感到高兴。”

——Chuck Allison, *The C++ Source* 编辑

“Steve 曾经教我学 C++。这要追溯到 1982 或 1983 年，我想他那时刚在贝尔实验室以实习生的身份和 Bjarne Stroustrup（C++ 的发明者）共事过一段时间。Steve 是未引起过多关注的早期英雄之一，他的任何作品我都优先列入阅读计划。本书是对 Steve 大量知识与经验的汇总，易于阅读，我强烈推荐！”

——Stan Lippman, *C++ Primer* (第 4 版) 作者之一

“作者有意使本书成为一本短小精悍的非傻瓜书，我喜欢这种风格。”

——Matthew P. Johnson, 哥伦比亚大学

“我赞同（作者）对不同类型的程序员的评价。作为一个开发者，我就遇到过这些类型的程序员，这样的书有助于填补他们之间的知识鸿沟……我认为这本书是对其他书籍（比如 Scott Meyers 的 *Effective C++*）很好的补充。它以一种简练且易读的风格将一切知识展现于你的面前。”

——Moataz Kamel, 资深软件设计师，摩托罗拉加拿大公司

“Dewhurst 又写了一本佳作。本书应该是那些正在使用 C++（并自以为对 C++ 无所不知）的人们的必备读物。”

——Clovis Tondo, *C++ Primer Answer Book* 作者之一

译者序

尽管 C++ 越来越像是“专家专用”的语言，精通它需要付出极大的努力，然而并非每一个人都需要成为 C++ 语言专家。对于大多数人而言，学习 C++ 的目的是为了“致用”，而非研究语言本身。我们的精力应该放在有效地掌握编程必备知识上，以便能够胜任目标领域的软件开发。职业程序员往往更应该是“领域专家”而非“语言专家”。

本书提供了 C++ 程序员所必须具备的“常识”。这里所说的“常识”并非意味着简单或平庸，而是指“必不可少”，事实上，有些内容相当高级，比如设计模式和泛型编程。本书对散布于许多其他 C++ 书籍中的知识进行了精心挑选，最终浓缩成 63 条。每一条款相对独立，可以随机查阅。许多条款内部还含有交叉索引，便于加深对该主题的理解。

这些条款涉及的主题范围较广，除了指针操作、面向对象、异常处理以及内存分配等主题外，对于现代 C++ 编程技术亦有很好的描述，其中仅设计模式就占了好几个条款。除了一个总论性的条款外，另外还具体介绍了 Prototype、Factory Method、Command、Template Method 等经典模式，而 trait、policy 以及智能指针亦可归入这个范畴。作者“短、平、快”式的介绍，可以使你迅速掌握这些常用模式的概念和用法。

现代 C++ 程序员应该像熟悉面向对象编程那样熟悉模板和泛型编程。本书中，模板和泛型编程内容占了条款总数的近 1/3，其中包括：类模板显式特化、模板局部特化、类模板成员特化、成员模板、嵌入的类型信息、模板的模板参数、模板实参推导、重载函数模板等等。这些都是进行泛型编程不可或缺的知识。

作者 Steve Dewhurst 是贝尔实验室 C++ 元老之一，有着 20 多年 C++ 应用经验，所解决的问题涉及多个领域，并是两款 C++ 编译器的作者。他的文风一向简练明快，并不失尖锐。阅读本书，可以给你带来“拨开迷雾见青天”的感觉。

本着对国内 C++ 应用现状的了解，我认为本书首先适合业界程序员用作快速参考。一些程序员项目经验不少，但对 C++ 的使用仅限于一小套子集，而且往往是一套原始的子集。这本小册子可以快速弥补这方面的知识结构缺陷。作者奉行“有所为、有所不为”的指导思想，忽略了复杂而很少使用的细节，便于直奔主题，抓住重点。此外，有了实

战经验作为后盾，本书很容易上手。

对于已经系统地学习过一门 C++ 课程的在校大学生来说，这本小书可以开阔你的眼界。如果你的 C++ 基础尚不足以顺畅地阅读书中部分条款，读来如雾里看花，或因作者点到为止而感觉意犹未尽，可以考虑选读“参考书目”中列出的书籍，它们大都有高品质的中文版。

在沉寂许久之后，一批优秀的 C++ 新作终于陆续面世，我有幸参与翻译包括 *C++ Common Knowledge*、*Imperfect C++* 以及 *C++ Template Metaprogramming* 等在内的几本佳作。希望在第一时间完成翻译的这本新书，能够给期待已久的你带来新鲜的快乐！

荣耀

2005 年 6 月

南京师范大学

royal@royaloo.com

前言

一本成功的书不是由书中的内容所构成的，而是由该书省略的内容所构成的。

——马克·吐温

……尽可能简单，但不过分简单。

——阿尔伯特·爱因斯坦

……一个对读者的能力持怀疑态度的作家根本不能称其为作家，只不过是个阴谋家而已。

——E. B. 怀特

当 Herb Sutter 接手 *C++ Report* 的编辑工作时，他很快就邀我为之写一个专栏，主题由我来定。我将该专栏命名为“Common Knowledge（常识）”。用 Herb 的话来说，该专栏预期为“对每一位职业 C++ 程序员应该知道但未必总是知道的基础知识之定期概述”。然而，在以那样的风格写了一些专栏文章后，我对模板元编程（*template metaprogramming*）技术的兴趣日渐浓厚，故而此后“Common Knowledge”中讨论的一些主题距离“Common”越来越远。

然而，在 C++ 程序设计界，当初促使我选定写这个专栏的问题仍然存在。在我的培训和咨询工作中，常常会遇到下面几类人员：

- 领域专家，他们是专家级的 C 程序员，但对 C++ 只有一些基本的认知（并可能对 C++ 怀有敌意）。
- 直接从大学雇来的新手，他们有才干，但对 C++ 语言只有理论上的认识，缺乏 C++ 产品开发经验。
- 专家级的 Java 程序员，他们仅有少量的 C++ 经验，并有以 Java 编程的方式来从事 C++ 编程的倾向。
- C++ 程序员，他们具有若干年维护现有 C++ 应用程序的经验，但尚未经受学习超出维护程序所需基础知识的任何东西之挑战。

我希望能即刻进行建设性的工作，但是，许多我共事过的或培训过的人在有能力解决

业务之前，都需要接受形形色色的关于 C++ 语言特性、模式以及编程技术的预备性的教育。更糟糕的是，我怀疑大多数 C++ 代码都至少忽略了其中一些基本要素，因而不具备大多数 C++ 专家所认可的产品级的质量。

这本书致力于解决这个具有普遍性的问题，它提供了每一位职业 C++ 程序员需要知道的常识，并且这些常识都被精简至本质，因此可被高效而精确地吸收。其中有不少信息也可以从其他途径获得，而有些知识则是所有专家级 C++ 程序员知道但未成文信息的完整摘要。本书优势在于，这些材料现在被集中于一处，并依据我多年的培训和咨询经验进行了遴选，经验表明，这些都是最常被误解同时也是最有用的语言特性、概念和技术。

也许构成本书的 63 个简短条款最重要的方面在于它们所省略掉的东西，而不是它们所包含的东西。许多主题都可以进行更复杂的讨论。一般作者对这些复杂性的忽略会导致传达的信息不够充分，从而可能会误导读者，但对一个主题的全部复杂性进行专家级的讨论，又可能会使读者淹没于知识的海洋之中。本书采取的方式是在讨论每一个主题时过滤掉那些“不必要”的复杂性。我希望有幸留下的这些东西是对产品级 C++ 编程所必需的知识的清晰萃取。较真的 C++ 语言专家可能意识到我省略了对一些有趣的、甚至重要的问题（从理论的角度来说）的讨论，但我所省略的那些东西通常并不会影响阅读和编写产品级 C++ 代码的能力。

写作这本书的另一个动机来自于我在一次会议上同一群知名 C++ 专家的谈话。这些专家对于一件事情颇感沮丧，那就是他们认为现代 C++ 是如此复杂，以至于“普通”程序员已经不再能够理解它了（比如，在模板和名字空间上下文中的名字绑定问题。是的，解决这样的问题确实需要普通 C++ 程序员下更多的功夫）。在我看来，应该说其实我们的态度有些过于自负了，我们的沮丧也是不合情理的。我们这些“专家”们自己就不存在这样的问题，实际上，使用 C++ 编程就像说一门（远比 C++ 复杂的）自然语言那样容易，尽管我们不能完全分析我们所说的每一句话的语法结构。本书不断出现的一个主题是，虽然对特定语言特性细节的完整描述可能让人望而生畏，但是，日常使用的语言特性都是直观而自然的。

不妨考虑一下函数重载。有关它的完整描述占据了很大一块标准文档，并占据了许多 C++ 教程的一整章（甚至多章）。然而，当我们面对如下代码时

```
void f( int );  
void f( const char * );  
//...  
f( "Hello" );
```

一个职业 C++ 程序员是不可能不知道哪一个 f 被调用的。有关重载函数调用解析规则的完整知识当然是有意义的，但很少需要用到。同样的道理适用于其他许多看上去很复杂

的 C++ 语言特性和惯用法。

这并不是说所有展示于本书中的材料都很简单，它们“尽可能简单，但不过分简单”。在 C++ 编程中，以及任何其他值得从事的智力活动中，许多重要的细节都无法写在“索引卡片”上。此外，这不是一本“傻瓜”书。我对自己对那些挤出宝贵时间来阅读我的书的读者负有极大的责任。我尊重这些读者，并且努力与他们交流，就像我亲自与同事交流一样。我认为给职业人员看八级难度的东西算不上写作，不过是想迎合少数人的胃口而已。

本书中的许多条款描述的是一些简单的误解，这些误解都是我曾一再看见的，只要予以指正即可（例如成员函数查找的作用域顺序、重写（`override`）和重载（`overload`）之间的区别等）。另外一些条款则论述那些正在成为职业 C++ 程序员所必需、但常常又被错误地认为过于困难并因而被避免使用的知识（例如类模板局部特化（`template partial specialization`）和模板的模板参数（`template template parameter`）等）。为此我受到了一些专家级审稿人的批评，说我在模板问题上花费的篇幅过多（约占全书的 1/3），而这些知识并非真的是“常识”。然而，这其中每一位专家又都指出有一两个甚至好几个模板主题应该包含于本书之中。一个有趣的现象是，这些建议中几乎没有重叠，每一个和模板有关的条款都至少有一个支持者。

这就是构成本书所含条款的问题症结。我并不认为有哪一位读者对本书每一个条款所谈的主题都一无所知，而且我还认为甚至有人熟悉本书中的所有条款。显然，如果某一位读者对某个特定的主题不熟悉，我认为阅读本书应该会从中受益。然而，即使某一位读者已经熟悉某个主题，我还是希望他能从一个全新的角度来阅读它，这样也许能够澄清一些轻微的误解或进一步加深对该主题的理解。这本书可能还有一个作用，那就是可以节省更多有经验的 C++ 程序员的宝贵时间。那些能干的 C++ 程序员常常发现他们一再被问及同样的问题，从而影响了他们自身的工作。希望“先读一读《C++ 必知必会》，再来和我讨论该问题”的回答方式，能够为这些 C++ 专家节省大量的时间，从而允许他们将自己的专家经验用在更复杂的问题上，而这些问题才是需要专家来解决的。

起先我试图将这 63 个条款分组到若干章中，那样显得更加整洁，但这些条款自己却并不这么认为。它们倾向于彼此簇拥在一起，就这一点而言，有些很明显，有些则有点出乎意料。举个例子，与异常和资源管理有关的条款形成了相当自然的一个组。而不那么明显的是，“能力查询”、“指针比较的含义”、“虚构造函数与 Prototype 模式”、“Factory Method 模式”以及“协变返回类型”这几个条款之间的关系紧密得有点出乎意料，因而最好被安排在一起。“指针算术”和“智能指针”放在一块儿，而不是和出现于本书较早部分的指针和数组方面的素材放在一起。因此，我不再试图将武断的章式结构强加于这些自然的分组上，而是决定授予个体条款自由结合权。自然而然，很多条款涉及的主题之

间存在许多其他相互关系，简单的线形顺序很难表达出这一点，因此条款中还频繁出现内部交叉引用。所以说，它们是一个簇拥但紧密连接的共同体。

尽管本书写作的主要指导思想是短小精悍，但对一个主题的讨论有时会包括一些辅助性的细节，尽管它们和眼前讨论的主题并不直接相关。对于该主题的讨论来说，这些细节并非必需，但读者由此可注意到特定程序或技术的存在。例如，在好几个条款中都出现的Heap模板例子可以让读者顺便了解到有用但很少被讨论到的STL堆算法，而对placement new的讨论则勾画出许多标准库组件所用到的复杂老练的内存管理技术基础。只要这么做看起来很自然，我就会利用机会，将辅助性话题的讨论混入某个特定的具名条款中。因此，条款“RAII”包含了对构造函数和析构函数激活顺序的简短讨论，条款“模板实参推导”讨论了用于特化类模板的辅助函数的使用，条款“赋值和初始化并不相同”则混入了对计算性的构造函数的讨论。这本书的条款数目很容易翻倍，但是，就像这些簇拥的条款自身一样，辅助性话题和具体条款的相关性，使得该主题被放置于合适的上下文之中，并且有助于读者高效、精确地吸收所表达的知识。

我很不情愿地包含了几个不适合在本书中讨论的主题（要知道，本书的写作风格是条款简短）。特别是有关设计模式和标准模板库的设计方面的主题，看上去短得可笑，很不完整。它们的现身只是为了消除一些常见的误解，并强调这些主题的重要性，从而鼓励读者去学习有关该主题更多的东西。

就像团聚在一起度假的家庭成员交换各自的趣事一样，提供例子早已成为我们编程文化的一个组成部分，因此Shape、String、Stack以及任何其他常见的“嫌犯”都一一露面。对这些基准例子达成共识，可以使我们的交流像使用设计模式那样高效。例如“设想我希望rotate（旋转）一个Shape，除了……之外”，或者“当连接两个String时……”这些常见的例子更适合于交流，可以避免费时的背景介绍，比如“你知道当你的兄弟被逮捕时的表现吗？呃，前些天……”

有别于我以前写的书，本书试图避免传递对一些糟糕的编程实践以及对C++语言特性误用的评判——那是其他一些书的目标，其中最好的一些书已被我列于“参考书目”之中（不过，我并没有完全成功地避免说教的倾向，本书中还是提到了一些糟糕的编程实践，虽然只是顺带一提）。一句话，本书的目的在于以尽可能高效的方式告诉读者产品级C++编程所需的本质技术。

Stephen C. Dewhurst

马萨诸塞州 卡沃尔

2005年1月

致谢

在忍受了我对 C++ 社群教育现状满腹牢骚很长一段时间后，Peter Gordon（一位非凡的人）建议我为之做一些实事，本书正是该建议的结果。Kim Boedigheimer 设法保持本书的一切事务顺利进行，使我未曾受到哪怕一次与写作有关的“威胁”。

感谢专家级技术审稿人 Matthew Johnson、Moataz Kamel、Dan Saks、Clovis Tondo 以及 Matthew Wilson，他们指出了手稿中的一些错误和语言表达不当之处，从而使本书变得更出色。作为一个老顽固，我并没有完全采纳他们的建议，因此，书中残存的任何错误和不妥，全是我的错。

本书中的一些材料曾出现于我在 *C/C++ Users Journal* 上开设的“Common Knowledge”专栏中，二者之间只有些细微的差别。另有不少材料曾经出现于 semantics.org 上的“Once, Weakly”Web 专栏中。我曾收到一些人士就印刷版和网络版文章提出的富有洞察力的评论，他们是 Chuck Allison、Attila Feher、Kevlin Henney、Thorsten Ottosen、Dan Saks、Terje Slettebo、Herb Sutter 以及 Leor Zolman。几次三番与 Dan Saks 的深入讨论使我对模板的特化和实例化之间的区别有了更深刻的理解，并帮助我澄清了普通重载与 ADL（Argument Dependent Lookup，实参相依的查找）以及“中缀操作符查找（infix operator lookup）”中出现的重载的区别。

本书同时还依赖于一些间接的贡献。非常感谢 Brandon Goldfedder 对出现于本书“设计模式”条款中的算法类比建议。感谢 Clovis Tondo 的激励以及在寻找优秀的审稿人方面给予的协助。我很幸运连续几年教授基于 Scott Meyers 的 *Effective C++*、*More Effective C++* 和 *Effective STL* 3 本书的课程。这使我获得了第一手的资料，让我认识到那些期望从符合业界标准的、中等难度的 C++ 书籍中获益的学生们通常欠缺哪些背景知识，这些观察帮助我确定了本书所要讨论的主题。Andrei Alexandrescu 的作品鼓舞我对模板元编程进行实验，而不是仅凭主观臆断行事。Herb Sutter 和 Jack Reeves 在异常方面的工作帮助我更好地理解应该如何使用异常。

我还要感谢邻居好友 Dick 和 Judy Ward，他们定期把我从计算机面前叫走，让我参加当地的蔓越桔收获活动。对于那些职业工作主要是处理对现实世界的简单抽象的人来说，这真是极有益于身心的调剂。从某种意义上讲，说服蔓越桔结果实和尝试模板局部特化这

二者之间的复杂性还真是有一拼。

一如既往，Sarah G. Hewins 和 David R. Dewhurst 为本书的写作提供了颇有价值的协助（当然也设置了一些必要的“障碍”）。

我自认为是一个性格沉稳的人，惯于自省而讨厌乌鸦般的喋喋不休。然而，就像有些人一旦手握方向盘就跟换了个人似的，我在写作本书手稿时差不多就是这个状态。Addison-Wesley 厉害的“行为矫正专家”们识破了我性格上的缺点。Chanda Leary Coutu 与 Peter Gordon、Kim Boedigheimer 齐心协力将我的手稿从感性的宣泄转换为理性的商业提议并监管实施。Molly Sharp 和 Julie Nahil 不仅将笨拙的 Word 文档变成现在看到的优美的页面，还改正了手稿中存在的诸多错误，并且保留了我那古老的句式结构、非同寻常的措辞以及特殊的连字符^①等写作风格。尽管我的要求变个不停，但 Richard Evans 还是设法使得本书如期出版，并且制作了两份单独的索引。Chuti Prasertsith 则为本书设计了精美的蔓越桔封面。多谢诸位！

^① 作者对连字符的用法比较怪异，不过你从中文版里看不出这一点。此外，不同于其他 C++ 书籍的是，本书还提供了一份“代码示例索引”。——译者注

目录

| | |
|-----------------------------|----|
| 条款 1 数据抽象 | 1 |
| 条款 2 多态 | 2 |
| 条款 3 设计模式 | 5 |
| 条款 4 STL | 8 |
| 条款 5 引用是别名而非指针 | 10 |
| 条款 6 数组形参 | 13 |
| 条款 7 常量指针与指向常量的指针 | 16 |
| 条款 8 指向指针的指针 | 19 |
| 条款 9 新式转型操作符 | 21 |
| 条款 10 常量成员函数的含义 | 25 |
| 条款 11 编译器会在类中放东西 | 29 |
| 条款 12 赋值和初始化并不相同 | 31 |
| 条款 13 复制操作 | 34 |
| 条款 14 函数指针 | 37 |
| 条款 15 指向类成员的指针并非指针 | 40 |
| 条款 16 指向成员函数的指针并非指针 | 43 |
| 条款 17 处理函数和数组声明 | 46 |
| 条款 18 函数对象 | 48 |
| 条款 19 Command模式与好莱坞法则 | 52 |

| | | |
|-------|--------------------------|-----|
| 条款 20 | STL函数对象 | 55 |
| 条款 21 | 重载与重写并不相同 | 58 |
| 条款 22 | Template Method模式 | 60 |
| 条款 23 | 名字空间 | 62 |
| 条款 24 | 成员函数查找 | 66 |
| 条款 25 | 实参相依的查找 | 68 |
| 条款 26 | 操作符函数查找 | 70 |
| 条款 27 | 能力查询 | 72 |
| 条款 28 | 指针比较的含义 | 75 |
| 条款 29 | 虚构造函数与Prototype 模式 | 77 |
| 条款 30 | Factory Method模式 | 79 |
| 条款 31 | 协变返回类型 | 82 |
| 条款 32 | 禁止复制 | 85 |
| 条款 33 | 制造抽象基类 | 86 |
| 条款 34 | 禁止或强制使用堆分配 | 88 |
| 条款 35 | placement new | 90 |
| 条款 36 | 特定于类的内存管理 | 93 |
| 条款 37 | 数组分配 | 97 |
| 条款 38 | 异常安全公理 | 100 |
| 条款 39 | 异常安全的函数 | 103 |
| 条款 40 | RAII | 106 |
| 条款 41 | new、构造函数和异常 | 110 |
| 条款 42 | 智能指针 | 112 |
| 条款 43 | auto_ptr 非同寻常 | 114 |
| 条款 44 | 指针算术 | 116 |

| | | |
|--------|-------------------------------------|-----|
| 条款 45 | 模板术语 | 119 |
| 条款 46 | 类模板显式特化 | 121 |
| 条款 47 | 模板局部特化 | 125 |
| 条款 48 | 类模板成员特化 | 129 |
| 条款 49 | 采用 <code>typename</code> 消除歧义 | 132 |
| 条款 50 | 成员模板 | 136 |
| 条款 51 | 采用 <code>template</code> 消除歧义 | 140 |
| 条款 52 | 针对类型信息的特化 | 142 |
| 条款 53 | 嵌入的类型信息 | 146 |
| 条款 54 | <code>traits</code> | 149 |
| 条款 55 | 模板的模板参数 | 154 |
| 条款 56 | <code>policy</code> | 159 |
| 条款 57 | 模板实参推导 | 163 |
| 条款 58 | 重载函数模板 | 167 |
| 条款 59 | <code>SFINAE</code> | 169 |
| 条款 60 | 泛型算法 | 172 |
| 条款 61 | 只实例化要用的东西 | 176 |
| 条款 62 | 包含哨位 | 179 |
| 条款 63 | 可选的关键字 | 181 |
| 参考文献 | | 184 |
| 索引 | | 185 |
| 代码示例索引 | | 195 |

条款 1

数据抽象

“类型”是一组操作，“抽象数据类型”则是一组具有某种实现的操作。当我们在某个问题领域中识别对象时，首先考虑的问题是“可以用这个对象来做什么”而不是“这个对象是如何实现的”。因此，如果某个问题的自然描述涉及到雇员、合同和薪水记录，那么用来解决该问题的编程语言就应该包含 `Employee`、`Contract` 和 `PayrollRecord` 类型。这样就允许在问题领域和解决方案领域之间进行双向、高效地转换，用这种方式编写的软件才能尽量避免产生“转换噪音”，从而达到更简洁、更准确。

在 C++ 这样的通用编程语言中，不会有像 `Employee` 这样特定于应用程序的类型，我们有更好的东西：C++ 为创建复杂的抽象数据类型提供了便利。从本质上说，抽象数据类型的用途在于将编程语言扩展到一个特定的问题领域。

C++ 中不存在针对抽象数据类型设计的公认方案，这方面的编程依然需要灵感和艺术才能，不过许多成功的途径都遵循下面这组类似的步骤。

(1) 为类型选择一个描述性的名字。如果难以为这个类型命名，那就说明你还不知道你想要实现什么，你需要多开动脑筋。一个抽象数据类型应该表示一个单纯的、有着良好定义的概念，而且为该概念所取的名字应该是显而易见的。

(2) 列出类型所能执行的操作。定义一个抽象数据类型的依据是能用它做什么。不要忘了初始化（构造函数）、清理（析构函数）、复制（复制操作）以及转换（不带 `explicit` 关键字修饰的单参数构造函数和转换操作符）。要避免在实现时简单地数据成员提供一串 `get/set`（获取/设置）操作——那不叫数据抽象，而是懒惰且缺乏想象力的表现。

(3) 为类型设计接口。正如 **Scott Meyers** 告诉我们的那样，一个类型应该做到“易于正确使用、难以错误使用”。既然抽象数据类型是对语言的扩展，那么务必要正确地进行语言设计。你要为类型的用户设身处地地想一想，并且编写一些使用类型接口的代码。良好的接口设计除需考虑技术的威力外，心理学和情感方面的问题同样需要加以考虑。

(4) 实现类型。不要让实现影响类型的接口。要实现类型的接口所承诺的约定。记住，在大多数情况下，对抽象数据类型的实现的改动，远比对其接口的改动来得频繁。

1

2

条款 2

多态

多态 (polymorphism) 在一些编程教程中被弄得很神秘, 而在另外一些教程中则被忽略, 其实它不过是 C++ 语言所支持的一个简单而有用的概念。按照 C++ 标准所言, “多态类型 (polymorphic type)” 就是带有虚函数的类类型 (class type)。从设计的角度来看, “多态对象 (polymorphic object)” 就是一个具有不止一种类型的对象, 而 “多态基类 (polymorphic base class)” 则是一个为满足多态对象的使用需求而设计的基类。

让我们来看一个金融期权的类型 AmOption, 如图 1 所示。

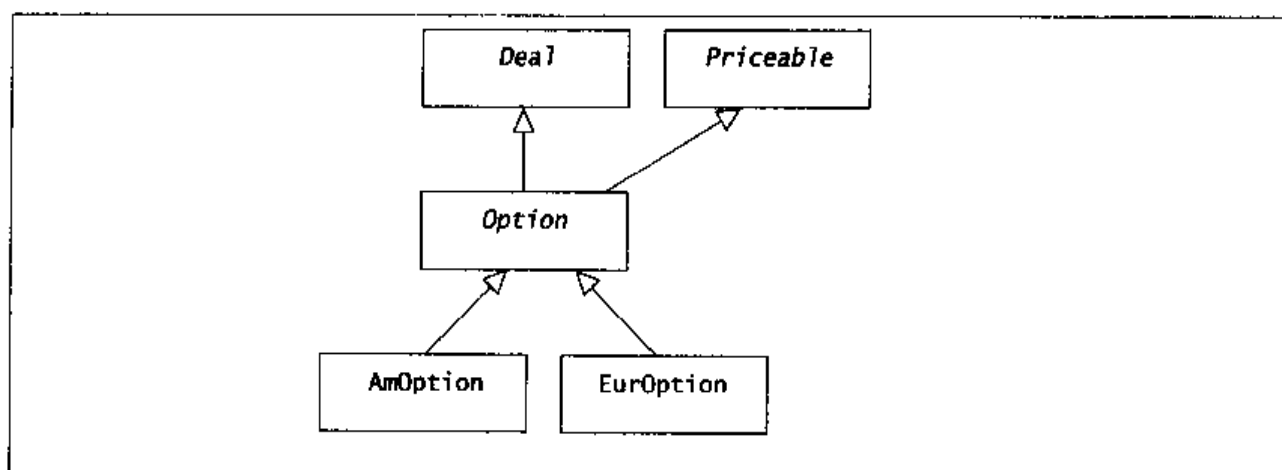


图 1 在一个金融期权层次结构中多态的作用。AmOption 有四种类型

AmOption 对象同时具有四种类型: AmOption、Option、Deal 以及 Priceable。由于一个类型是一组操作 (参见 “数据抽象[条款 1]” 和 “能力查询[条款 27]”), 因此, AmOption 对象可以通过其 4 个接口中的任何一个进行操纵。这意味着一个 AmOption 对象可以被针对 Deal、Priceable 和 Option 接口编写的代码所操纵, 从而允许 AmOption 的实现利用或复用所有那些代码。对于 AmOption 这样的多态类型, 从基类继承的最重要的东西就是它们的接口, 而不是它们的实现。事实上, 一个基类仅仅由接口组成不但常见, 而且通常正是我们所希望的 (参见 “能力查询[条款 27]”)。

当然, 这里有一个需要注意的地方。如果让这种优势能够发挥出来, 一个良好设计的

多态类对于它的每个基类而言必须是可替换的。换句话说，如果针对 Option 接口编写的通用代码接受的是一个 AmOption 对象，那么该对象的行为最好就像一个 Option 对象！

这并不是说 AmOption 对象应该和 Option 对象的行为完全一致（首先可能是因为 Option 基类的许多操作是不带任何实现的纯虚函数）。实际上，将一个多态基类（如 Option）想象成一份契约更好理解一些。这个基类对其接口的用户做了某些承诺，这些承诺包括郑重的语法承诺，即特定的成员函数可以通过一些特定类型的实参进行调用，以及不太容易验证的语义上的承诺，即当一个特定的成员函数被调用时将会发生什么实际情况。像 AmOption 和 EurOption 这样的具体派生类被称为“转包者”，它们实现 Option 与其客户签订的契约，如图 2 所示。

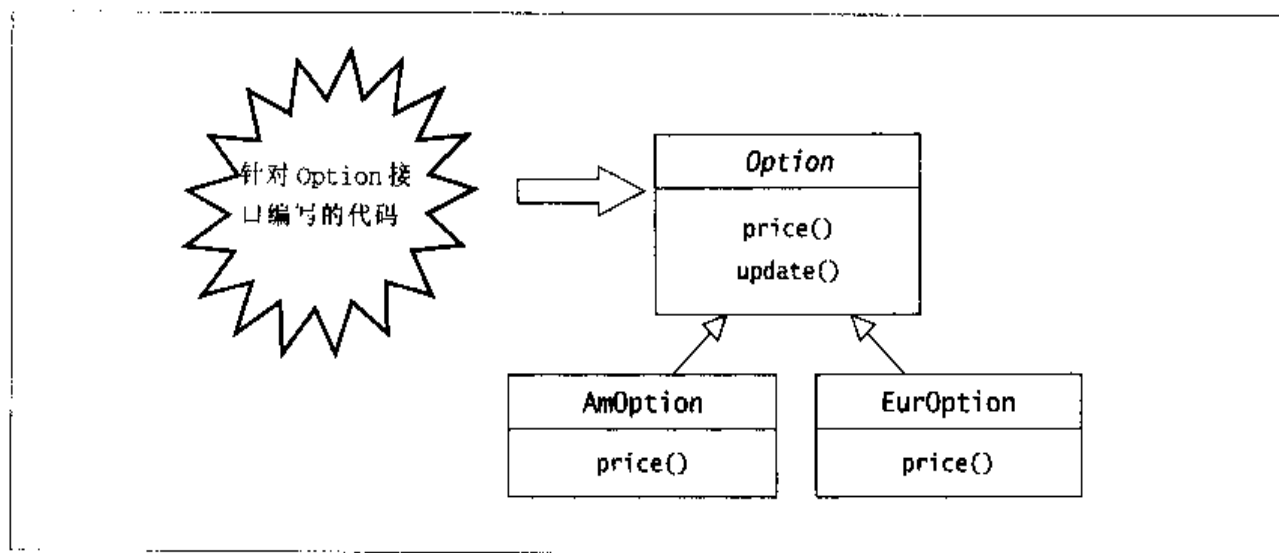


图2 一个多态的承包者及其“转包者”。基类 Option 指定了一个契约

举个例子，如果 Option 具有一个纯虚成员函数 price，其作用是给出 Option 的当前值，那么 AmOption 和 EurOption 都必须实现这个函数。我们显然不会为这两种类型的 Option 实现完全一致的行为，但它们都应该计算并返回一个价格（price），而不应该去拨打一个电话或打印一份文件。

4

另一方面，如果我要去访问同一个对象的两种不同接口的 price 函数，那么我应该得到相同的结果。就本质而言，每一个调用都应该绑定到同一个函数：

```
AmOption *d = new AmOption;
Option *b = d;
d->price(); // 如果这一个调用的是 AmOption::price……
b->price(); // ……那么这一个也应该如此！
```

这是有意义的（在高级面向对象编程中，竟然有如此之多的基本常识被费解的语法所掩盖）。假如我问你“那个美国期权的当前值是什么？”，我期望得到与以下简短提问方式相同的答案：“那个期权的当前值是什么？”

当然，同样的推理也适用于对象的非虚拟函数：

```
b->update(); // 如果这一个调用的是 Option::update.....  
d->update(); // .....那么这一个也是如此！
```

正是基类提供的契约允许针对基类接口编写的“多态”代码对特定的期权起作用，同时有助于对派生类的存在保持“健康的不知情”。换句话说，多态代码可能正在操纵 AmOption 和 EurOption 对象，但除非特别关心它们到底是什么对象，否则均被视作 Option 对象。各种各样“具体的”Option 类型可以被添加或删除而不会影响到只关心基类 Option 的通用代码。比方说，如果在某一个地方出现一个 AsianOption 对象，那么只知道 Option 的多态代码也能够操作它，这全托“对其具体类型不知情”的福，如果以后这个对象消失了，也犯不着去挂念它。

出于同样的原因，像 AmOption 和 EurOption 这样具体的期权类型只需要知道基类就可以了（它们实现了基类的契约），改变通用代码对它们毫无影响。原则上，基类可以不知道除自身以外的任何事物。从实践的角度来看，对其接口的设计要考虑预期用户的需求，并且应该以这样的方式进行设计：派生类可以很容易地推知并实现其契约（参见“Template Method 模式[条款 22]”）。然而，基类应该对其派生类的具体细节全然不知，因为知道这些会不可避免地致使在类层次结构上添加或删除派生类变得困难。

和生活中一样，在面向对象设计中，“不知情”或“忽略”也是天赐之福（参见 [5] “虚构造函数与 Prototype 模式[条款 29]”和“Factory Method 模式[条款 30]”）。

条款 3

设计模式

对于任何还不熟悉设计模式的人来说，在对这个领域进行简短的纵览之后，可能会留下这样的印象：设计模式是一个市场营销大骗局，它不过是一些简单的编程技术，或者不过是计算机科学家（这些科学家没事应该多出来走走）的玩物。尽管这些印象都有那么一点道理，然而设计模式的确是职业 C++ 程序员工具箱中不可或缺的组件。

设计模式是一个被反复谈论的架构主题，它为特定上下文中的常见设计问题提供了解决方案，并描述了这种解决方案的结果。设计模式不仅仅是对技术的简单描述，它还是从现有的成功实践一点一滴汇集起来的设计智慧的具名封装，并以容易交流和复用的方式编写而成。模式关乎程序员之间的顺畅交流。

从实践的角度来看，设计模式具有两个重要的属性。首先，它们描述了经过验证的、成功的设计技术，这些技术可以按上下文相关的方式进行定制，以便满足新的设计场合的要求。其次，并且可能更重要的是，在提及某个特定模式的应用时不仅包括其中用到的技术，还包括应用该模式的动因以及应用后所达到的效果。

这类事情并不是什么新东西。考虑一个来自算法领域的类比（需要说明的是，算法不是设计模式，也不是“编程模式”，它们只是算法，这里不过是一个类比而已）。考虑（我可能对一位同事作的）如下声明：“有一个未排序的序列，必须要进行很多次搜索。因此，希望对其进行快速排序，并且使用二分查找来执行每一个查找。”能够使用术语“快速排序”和“二分查找”，这种价值是不可估量的，不但在设计方面如此，在就该设计与受过教育的同事进行交流时也是如此。当我说“快速排序”时，我的同事知道我正在排序的序列具有随机存取结构，并且它的排序时间复杂度为 $O(n \log_2 n)$ ，同时该序列中的元素可以采用类似小于操作符进行比较。当我说“二分查找”时，我的同事知道（即使事先没有提到“快速排序”）序列已经被排序过了，定位感兴趣的元素所执行的比较操作的时间复杂度为 $O(\log_2 n)$ ，并且存在一个适当的操作来对序列中的元素进行比较。标准算法所具有的共享的知识以及标准词汇表，不但允许高效地记录文档，而且允许对设计方案进行有效的评议。比方说，如果我计划对一个单向链接表结构来执行这个查找和排序过程，我的同事立刻会自鸣得意地哈哈大笑，并指出在这种情形下我不能使用快

速排序并且可能不希望使用二分查找。

在设计模式出现之前，在对面向对象设计的文档化、交流以及高效地评议方面，这些优点都不具备。我们被迫低水平地描述我们的设计，这种方式低效且不够精确。这并不是说用于复杂面向对象设计的技术尚不存在，而是这些技术尚未以一个共享的、通用术语的方式为整个编程社群所用。设计模式解决了这个问题，我们现在可以像描述算法设计一样高效、毫无歧义地描述面向对象设计。

举个例子，当我们看到 **Bridge** 模式被应用到某项设计中时，我们知道在一个简单的机制层面，抽象数据类型实现被分离成一个接口类和一个实现类。此外，我们知道这样做的原因是为了将接口从实现强有力地分离出来，这样，对实现的改变将不会影响到使用接口的用户。我们还知道这种分离会带来运行期开销，知道应该怎样对抽象数据类型的源代码进行布局，还知道许多其他细节。模式的名字是关于某项技术的诸多信息和经验的高效且无歧义的“句柄”，在设计和文档中小心并正确地使用模式和模式术语，可以使代码和设计更加明晰。

那些严谨的模式专家有时以某种文献的形式来描述模式（他们确实是这么做的），这种描述遵循某种正式的结构。还有几种常见的变体也在使用，但不管哪种描述方式，均包含以下 4 个必不可少的部分。

首先，设计模式必须具有一个毫无歧义的名字。例如，术语“包装器 (wrapper)”对于设计模式命名来说就没有什么意义，因为它早已被广泛使用并且具有许多不同的含义。8 使用“**Wrapper**”这样的术语来命名某种设计模式只会带来混淆和误解。实际的做法是，以前在“wrapper”名下的设计技术现在分别被指派为“**Bridge**”、“**Strategy**”、“**Facade**”、“**Object Adapter**”以及其他一些模式名字。使用精确的模式名字比使用不那么精确的名字具有“明显”的优势，就像术语“二分查找”比“查找”更精确、更有意义一样。

其次，模式描述必须定义该模式所能解决的问题。这种描述可以相对宽泛，也可以相对狭窄。

再次，模式描述要记述该问题的解决方案。根据陈述的问题，该解决方案可以相对高级，也可以相对低级，但无论如何，它应该具有足够的通用性，以便可以根据问题可能出现的不同上下文进行定制。

最后，模式描述要记述将该模式应用于某个上下文的后果。在应用该模式后，该上下文是如何发生改变的？不管是变好，还是变坏。

拥有模式的知识可以使一名糟糕的设计师摇身一变成为一名优秀的设计师吗？呃，是给

出另一个类比的时候了：设想你被迫学习某一门让人痛苦的数学课，它的期末考核内容是证明某个数学领域中的许多定理。如何才能从这门课中死里逃生呢？当然，最显而易见的方式是成为一个天才。你从最初的原理开始，进而研究整个数学分支的基础知识，最终证明那些定理。一个更为实际的途径是，你牢记并消化该数学领域中的大量定理，并使用你所具备的任何天赋的数学能力、灵感以及好运去选择适当的辅助定理，然后以某种逻辑“胶水”将它们粘合在一起，从而最终证明新的定理。是的，甚至对于那些“传说中的天才”来说，这种方式都是很有优势的，因为基于现成的定理来证明会更高效，同“凡夫俗子”交流起来也更容易。当然，熟悉辅助定理并不能保证一个可怜的学数学的学生通过考试，但这类知识至少可以使其能够理解别人给出的证明。

同样的道理，从最初的原理进而到复杂的面向对象设计也是颇为无趣的，而且与他人交流最终设计也很困难。组合使用各种设计模式来生成面向对象设计，类似于在数学中使用辅助定理来证明一个新的定理。设计模式常常被描述为“微架构（micro-architecture）”，它们可以与其他模式进行组合从而生成一个新的架构。当然，选择适当的模式并有效地对其进行组合，也是需要设计方面的专家经验和天资禀赋的。不过，一旦设计完成后，甚至你的经理都能够理解完整的设计方案，只要他具备一些必需的模式方面的知识即可。

9

10

条款 4

STL

对 STL (Standard Template Library, 标准模板库) 的简短描述并不足以体现其设计上的过人之处。接下来的文字不过是鼓励你深入学习 STL 的“开胃小菜”。

STL 并不仅仅是一个库，它更是一种优秀的思想以及一套约定。

STL 包含三大组件：容器、算法和迭代器。容器用于容纳和组织元素；算法执行操作；迭代器则用于访问容器中的元素。这些都不是什么新东西，许多传统的程序库也都包含类似的组件，并且许多程序库也都是采用模板实现而成。STL 的优秀思想体现在：容器与在容器上执行的算法之间无需彼此了解，这种戏法是通过迭代器实现的。

迭代器类似于指针（实际上指针就是一种 STL 迭代器）。像指针一样，迭代器可以指向序列中的一个元素，也可以对其进行解引用（dereference），以便获得它所指向的对象的值。我们可以像操纵指针那样操纵迭代器，使其指向序列中不同的元素。STL 迭代器既可以是预定义的指针，也可以是用户自定义的类类型，当然，这种类型需要重载适当的操作符，以便与预定义指针拥有相同的使用语法（参见“智能指针[条款 42]”）。

STL 容器是对数据结构的一种抽象，以类模板的方式实现而成。由于具有不同的数据结构，因此不同的容器以不同的方式来组织其元素，以便对存取或操纵进行优化。STL 定义了 7 个（算上 string 则是 8 个）标准容器，另外，一些被广泛接受的非标准容器也得到了实现。

STL 算法是对函数的一种抽象，采用函数模板实现（参见“泛型算法[条款 60]”）。大多数 STL 算法用于处理一个或多个序列的值，其中每一个序列由一对有序的迭代器定义。其中第一个迭代器指向序列的第一个元素，第二个迭代器则指向序列最后一个元素之后的那个位置（而不是最后一个元素）。如果两个迭代器指向同一个位置，那么它们就定义了一个空序列。

11

迭代器提供了一种使容器与算法协同工作的机制。一个容器可以生成一对迭代器来指定一个元素序列（可以是全部元素，也可以只是一个子区间），而算法则对该序列进行操作。采用这种方式，容器和算法可以紧密地协作，同时还可以保持彼此不知情（这种

“不知情”的好处，乃是 C++ 高级编程领域反复强调的主题。参见“多态[条款 2]”、“Factory Method 模式[条款 30]”、“Command 模式与好莱坞法则[条款 19]”以及“泛型算法[条款 60]”）。

除了容器、算法和迭代器之外，STL 还定义了大量的辅助性功能。算法和容器可以采用函数指针和函数对象（参见“STL 函数对象[条款 20]”）根据需要进行定制，而这些函数对象又可以通过形形色色的函数对象适配器（function object adapter）进行配接和联合。

容器也可以利用容器适配器（container adapter）进行配接，从而将容器的接口修改为栈、队列或优先队列。

STL 对约定（convention）有着很强的依赖。容器和函数对象必须通过一套标准的嵌套类型名字对其自身进行描述（参见“嵌入的类型信息[条款 53]”、“traits [条款 54]”以及“STL 函数对象[条款 20]”）。容器和函数对象适配器均要求成员函数具有特定的名字并包含特定的类型信息。算法要求传递给它的迭代器支持特定的操作并能够识别是什么样的操作。当使用或扩展 STL 时，如果弃约定于不顾，那么同时也就放弃了美梦成真的希望。遵守约定，将拥有简单而轻松的生活。

STL 约定并未指明具体的实现细节，但对实现指定了效率方面的约束。此外，由于 STL 是一个模板库，许多优化和性能调整可以在编译期进行。前面提到的命名和信息约定对编译期优化具有显著的影响。一般而言，STL 的效率可以与专家手写代码的效率相媲美，而与普通程序员和程序员小组的手写代码相比则明显胜出一筹。另外，使用 STL 可以使代码变得更清晰，更易于维护。

学习 STL、并广泛地使用 STL 吧！

条款 5

引用是别名而非指针

引用（reference）是一个现有对象的别名。用对象来初始化引用之后，那么对象的名字或引用的名字都可以用于指向（refer to）该对象：

```
int a = 12;
int &ra = a; // ra 是 a 的别名
--ra; // a == 11
a = 10; // ra == 10
int *ip = &ra; // ip 指向 a
```

人们常常会将引用和指针混淆，原因大概在于 C++ 编译器通常采用指针的方式实现引用，但引用其实不是指针，其行为和指针并不相同。

在引用和指针之间存在三大区别：其一，不存在空引用（null reference）；其二，所有引用都要初始化；其三，一个引用永远指向用来对它初始化的那个对象。比如说，在先前的例子中，引用 ra 在整个生命期内都指向 a。绝大多数对引用的误用都滋生于对这三大区别的误解。

一些编译器可以捕捉到那些明显的创建空引用的尝试：

```
Employee &anEmployee = *static_cast<Employee*>(0); // 错误!
```

然而，编译器可能无法侦测到不那么明显的创建空引用的尝试，从而导致在运行期发生未定义行为：

```
Employee *getAnEmployee();
//...
Employee &anEmployee = *getAnEmployee(); // 可能是糟糕的代码
if( &anEmployee == 0 ) // 未定义的行为
```

13

如果 getAnEmployee 返回的是一个空指针，那么其后代码的行为就是未定义的。在这个例子中，最好使用一个指针来存放 getAnEmployee 返回的结果。

```
Employee *employee = getAnEmployee();
```

```
if( employee ) //...
```

引用必须初始化的要求，意味着当一个引用初始化时它所指向的那个对象必须存在。这一点很重要，因此我想再重复一遍：一个引用就是在该引用被初始化之前已经存在的一个对象的别名。一旦一个引用被初始化去指向一个特定的对象，那么该引用以后就不可以再指向别的对象；在一个引用的整个生命期内，该引用被绑定到用于初始化它的那个对象上。实际上，一个引用完成其初始化后，就只是初始化它的那个对象的别名了。这个“别名”属性使得引用常常成为函数形参的优秀选择。在以下 swap 函数模板中，形参 a 和 b 乃是传递给调用的实参的别名：

```
template <typename T>
void swap( T &a, T &b ){
    T temp(a);
    a = b;
    b = temp;
}
//...
int x = 1, y = 2;
swap( x, y ); // x == 2, y == 1
```

在以上对 swap 的整个调用期间，a 是 x 的别名，b 是 y 的别名。提醒一下，引用所指向的对象可以没有名字，因此引用可用于为没有名字的对象赋予一个方便的名字：

```
int grades[MAX];
//...
swap( grades[i], grades[j] );
```

当 swap 中的形参 a 和 b 分别被用实参 grades[i] 和 grades[j] 初始化以后，这两个没有名字的数组元素就可以通过别名 a 和 b 进行操纵了。为了简化和优化，还可以采用更直接的方式来使用这个属性。

14

考虑如下函数，它用于设置二维数组中一个特定元素：

```
inline void set_2d( float *a, int m, int i, int j ) {
    a[i*m+j] = a[i*m+j] * a[i*m+i] + a[i*m+j]; // 哎呀!
}
```

可以将注释有“哎呀！”的那行代码替代为更简单的版本，该版本利用了引用，而且还带来额外的好处，那就是正确！（看出错在哪儿了吗？反正我第一眼没看出来。）

```
inline void set_2d( float *a, int m, int i, int j ) {
    float &r = a[i*m+j];
    r = r * r + r;
}
```

一个指向非常量的引用是不可以用字面值或临时值进行初始化的：

```
double &d = 12.3; // 错误!  
swap( std::string("Hello"), std::string(", World") ); // 错误!
```

然而，一个指向常量的引用就可以：

```
const double &cd = 12.3; // OK  
template <typename T>  
T add( const T &a, const T &b ) {  
    return a + b;  
}  
//...  
const std::string &greeting  
    = add(std::string("Hello"),std::string(", World")); // OK
```

当一个指向常量的引用采用一个字面值来初始化时，该引用实际上被设置为指向“采用该字面值初始化”的一个临时位置。因此，`cd`并非真的指向字面值12.3，而是指向一个采用12.3初始化的、类型为`double`的临时变量。`greeting`引用则指向对`add`的调用所返回的无名临时`string`值。一般来说，这类临时对象在创建它们的表达式的末尾被销毁（确切地说，就是离开作用域并且析构函数被调用）。然而，当这类临时对象用于初始化一个指向常量的引用时，在引用指向它们期间，这些临时对象会一直存在。

条款 6

数组形参

数组形参是个容易出问题的地方。对于 C/C++ 新手而言，最大的惊讶是 C++ 中根本不存在所谓的“数组形参”，因为数组在传入时，实质上只传入指向其首元素的指针。

```
void average( int ary[12] ); // 形参ary 是一个 int *
//...
int anArray[] = { 1, 2, 3 }; // 一个具有 3 个元素的数组
const int anArraySize =
    sizeof(anArray)/sizeof(anArray[0]); // == 3
average( anArray ); // 合法!
```

这种从数组到指针的自动转换被赋予了一个迷人的技术术语：“退化”，即数组退化成指向其首元素的指针。顺便提及，同样的事情也发生在函数上。一个函数型参数会退化成一个函数指针。不过，和数组在退化时会丢失边界不同，一个退化的函数具有良好的感知能力，可以保持其参数类型和返回类型（同时还要注意对 anArraySize 所执行的适当的编译期计算，这种计算可以抗得住数组初始化元素的改变以及数组元素类型的改变）。

由于在数组形参中数组边界被忽略了，因此通常在声明时最好将其省略。然而，如果函数期望接受一个指向一个元素序列（换句话说，就是数组）的指针作为参数，而不是指向单个对象的指针，那么最好这样声明：

```
void average( int ary[] ); // 形参ary 仍然是一个 int *
```

另一方面，如果数组边界的精确数值非常重要，并且希望函数只接受含有特定数量的元素的数组，可以考虑使用一个引用形参：

```
void average( int (&ary)[12] );
```

现在函数就只能接受大小为 12 的整型数组：

```
average( anArray ); // 错误! anArray 是一个 int [3]!
```

模板有助于代码的泛化：

```
template <int n>
```

```
void average( int (&ary)[n] ); // 让编译器帮我们推导 n 的值
```

不过，更为传统的做法是将数组的大小明确地传入函数：

```
void average_n( int ary[], int size );
```

当然，我们可以将这两种方式结合起来：

```
template <int n>
inline void average( int (&ary)[n] )
    { average_n( ary, n ); }
```

从以上讨论中我们应该清晰地获知，使用数组作为函数参数最大的问题在于，数组的大小必须以形参的方式显式地编码，并以单独的实参传入，或者在数组内部以一个结束符值作为指示（例如用于指示“用作字符串的字符数组”之末尾的‘\0’）。另一个困难在于，不管数组是如何声明的，一个数组通常是通过指向其首元素的指针进行操纵的。如果那个指针作为实参被传递给函数，我们前面声明引用形参的技巧将无济于事。

```
int *anArray2 = new int[anArraySize];
//...
average( anArray2 ); // 错误！不可以使用 int * 初始化 int (&)[n]
average_n( anArray, anArraySize ); // 没问题
```

出于这些原因，经常采用某种标准容器（通常是 vector 或 string）来代替对数组的大多数传统的用法，并且通常应该优先考虑使用标准容器。另请参考“只实例化要用的东西[条款61]”中描述的 Array 类模板。

从本质上来说，多维数组形参并不比一维数组来得困难，但它们看上去更具挑战性：

```
void process( int ary[10][20] );
```

和一维数组的情形一样，形参不是一个数组，而是一个指向数组首元素的指针。不过，多维数组是数组的数组，因此形参是一个指向数组的指针（参见“处理函数和数组声明符[条款17]”和“指针算术[条款44]”）：

```
void process( int (*ary)[20] ); // 一个指针，指向一个具有 20 个 int 元素的数组
```

注意，第二个（以及后续的）边界没有退化，否则将无法对形参执行指针算术（参见“指针算术[条款44]”）。如前所述，让代码的读者清晰地知道你期望的实参是一个数组，这通常是一个好主意：

```
void process( int ary[][20] ); // 仍然是一个指针，但更清晰
```

对多维数组形参的有效处理往往“退化”成一个低级的编程练习，此时需要程序员取代编译器来执行索引计算：

```
void process_2d( int *a, int n, int m ) { // a是一个n×m的数组
    for( int i = 0; i < n; ++i )
        for( int j = 0; j < m; ++j )
            a[i*m+j] = 0; // 手工计算索引!
}
```

同样，有时模板有助于让事情更干净利落：

```
template <int n, int m>
inline void process( int (&ary)[n][m] )
{ process_2d( &ary[0][0], n, m ); }
```

一句话，数组形参是个讨厌的家伙，和它亲近你得小心。

条款 7

常量指针与指向常量的指针

在日常交流中，当一个 C++ 程序员说“常量指针 (const pointer)”时，其实他想表达的意思往往是“指向常量的指针 (pointer to const)”。真不幸，这是两个完全不同的概念。

```
T *pt = new T; // 一个指向 T 的指针
const T *pct = pt; // 一个指向 const T 的指针
T *const cpt = pt; // 一个 const 指针，指向 T
```

将 const 修饰符放到指针声明之前，应该想好，到底想叫什么东西变成常量，是指针？还是准备指向的那个对象？或兼而有之？在 pct 的声明中，指针不是 const 的，但它所指向的对象被认为是 const 的。换句话说，const 修饰符修饰的是基础类型 (base type) T 而不是指针修饰符 *。而对于 cpt 的声明来说，声明的是一个指向一个非常量对象的常量指针，即 const 修饰符修饰的是指针修饰符 * 而不是基础类型 T。

声明中的修饰符（即指针声明中第一个 * 修饰符之前出现的任何东西）的顺序无关性进一步混淆了围绕指针和常量的语法问题。例如，以下两行代码所声明的变量的类型完全相同：

```
const T *p1; // 一个指向常量 T 的指针
T const *p2; // 也是一个指向常量 T 的指针
```

第一种形式更传统一些，但如今许多 C++ 专家推荐使用第二种形式。理由在于，第二种形式不太容易被误解，因为这种声明可以倒过来读，即“指向常量 T 的指针”。使用哪一种形式无关紧要，只要保持一致就行了。然而，务必小心一个常见的错误，即将常量指针的声明与指向常量的指针的声明混淆。

```
T const *p3; // 一个指向常量的指针
T *const p4 = pt; // 一个常量指针，指向非常量 T
```

当然，可以声明一个指向常量的常量指针：

```
const T *const cpct1 = pt; // 二者均为常量
```

```
T const *const cpct2 = cpct1; // 同上
```

注意，使用一个引用通常比使用一个常量指针更简单：

```
const T &rct = *pt; // 而不是 const T *const
T &rt = *pt; // 而不是 T *const
```

注意到在前面的一些例子中，我们能够将一个指向非常量的指针转换为一个指向常量的指针。例如，我们能够使用 `pt`（类型为 `T *`）初始化 `pct`（类型为 `const T *`）。从非技术的角度来说，这样做之所以合法，是因为不会产生任何不良后果。考虑当一个非常量对象的地址被复制到一个指向常量的指针时的情形，如图3所示。

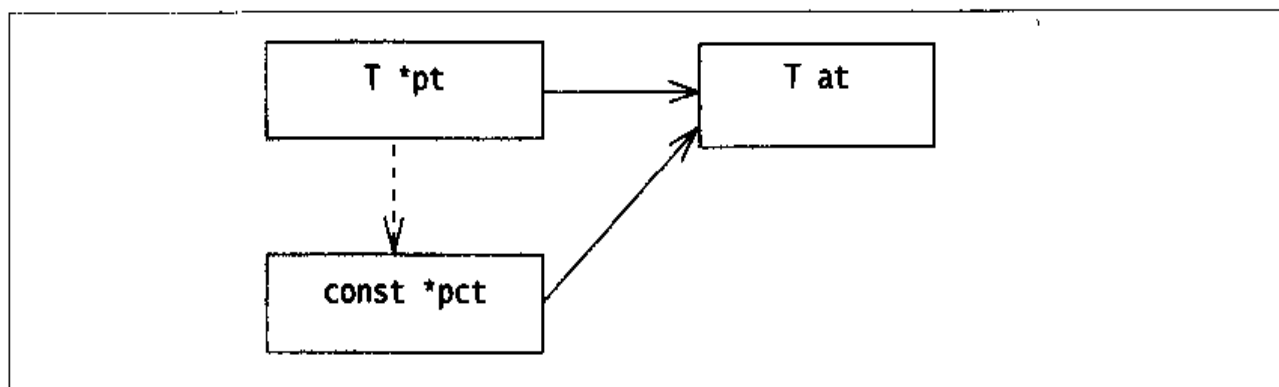


图3 一个指向常量的指针可以指向一个非常量对象

指向常量的指针 `pct` 现在指向一个非常量 `T`，但这不会造成任何危害。实际上，指向常量的指针（或引用）去指向非常量的对象，是司空见惯的事情：

```
void aFunc( const T *arg1, const T &arg2 );
//...
T *a = new T;
T b;
aFunc( a, b );
```

调用 `aFunc` 时，使用 `a` 初始化 `arg1`，使用 `b` 初始化 `arg2`。我们并没有宣称 `a` 要指向一个常量对象，或者 `b` 是一个常量引用，只是声明在 `aFunc` 函数中它们被视为常量，而不管它们实际上是否如此。这很有用，不是吗？

相反的转变，即从指向常量的指针转换为指向非常量的指针，则是非法的，因为可能会导致危险的后果，如图4所示。

在这个例子中，`pct` 可能实际上指向一个被定义为常量的对象。如果我们能够将一个指向常量的指针转换为一个指向非常量的指针，那么 `pt` 就可用于改变 `acT` 的值。

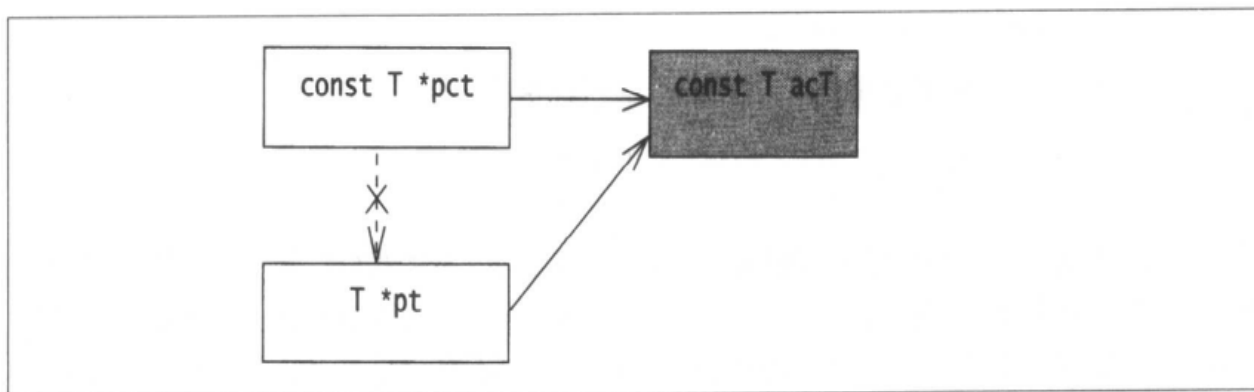


图4 指向非常量的指针不可以指向常量对象

```

const T acT;
pct = &acT;
pt = pct; // 报错! 很幸运……
*pt = acT; // 试图修改常量对象!

```

C++ 标准告诉我们，这样的赋值会导致未定义的结果，也就是说，我们不知道究竟会发生什么，不过可以肯定的是，不会发生什么好事。当然，我们可以利用 `const_cast` 显式地执行类型转换。

```

pt = const_cast<T*>(pct); // 没错，但这种做法不妥
*pt = acT; // 试图修改常量对象!

```

然而，如果 `pt` 指向一个被声明为常量的对象（例如 `acT`），那么以上赋值行为仍然是未定义的（参见“新式转型操作符[条款9]”）。

条款 8

指向指针的指针

声明指向指针的指针是合法的。这就是 C++ 标准所说的“多级”指针。

```
int *pi; // 一个指针
int **ppi; // 一个多级（2 级）指针
int ***pppi; // 一个多级（3 级）指针
```

尽管超过两级的多级指针很罕见，但在两种常见的情形下，确实会看到指向指针的指针。第一种情形是当我们声明一个指针数组时：

```
Shape *picture[MAX]; // 一个数组，其元素为指向 Shape 的指针
```

由于数组的名字会退化为指向其首元素的指针（参见“数组形参[条款 6]”），所以指针数组的名字也是一个指向指针的指针：

```
Shape **pic1 = picture;
```

我们在管理指针缓冲区的类的实现中最常看到这种用法：

```
template <typename T>
class PtrVector {
public:
    explicit PtrVector( size_t capacity )
        : buf_(new T *[capacity]), cap_(capacity), size_(0) {}
    //...
private:
    T **buf_; // 一个指针，指向一个数组，该数组元素为指向 T 的指针
    size_t cap_; // 容量
    size_t size_; // 大小
};
//...
PtrVector<Shape> pic2( MAX );
```

25

从 PtrVector 的实现可以看到，指向指针的指针可能会很复杂，最好将其隐藏起来。

多级指针的第二个常见应用情形，是当一个函数需要改变传递给它的指针的值时。考虑如下函数，它将一个指针移动到指向字符串中的下一个字符：

```
void scanTo( const char **p, char c ) {
    while( **p && **p != c )
        ++*p;
}
```

传递给 scanTo 的第一个参数是一个指向指针的指针，该指针值是我们希望改变的。这意味着我们必须传递指针的地址：

```
char s[] = "Hello, World!";
const char *cp = s;
scanTo( &cp, ',' ); // 将 cp 移动到第一个 “,” 出现的位置
```

这种用法在 C 中是合理的，但在 C++ 中，更习惯、更简单、更安全的做法是使用指向指针的引用作为函数参数，而不是使用指向指针的指针作为参数。

```
void scanTo( const char *&p, char c ) {
    while( *p && *p != c )
        ++p;
}
//...
char s[] = "Hello, World!";
const char *cp = s;
scanTo( cp, ',' );
```

在 C++ 中，几乎总是首选使用指向指针的引用作为函数参数，而不是指向指针的指针。

一个常见的误解是：适用于指针的转换同样适用于指向指针的指针。事实并非如此。例如，我们知道一个指向派生类的指针可被转换为一个指向其公共基类的指针：

```
Circle *c = new Circle;
Shape *s = c; // 挺好的……
```

26 因为 Circle 是一个 (is-a) Shape，因而一个指向 Circle 的指针也是一个 Shape 指针。然而，一个指向 Circle 指针的指针并不是一个指向 Shape 指针的指针：

```
Circle **cc = &c;
Shape **ss = cc; // 错误！
```

当涉及 const 时也会发生同样的混淆。我们知道，将一个指向非常量的指针转换为一个指向常量的指针是合法的（参见“常量指针与指向常量的指针[条款7]”），但不可以将一个指向“指向非常量的指针”的指针转换为一个指向“指向常量的指针”的指针：

```
char *s1 = 0;
const char *s2 = s1; // 没问题
char *a[MAX]; // 也就是 char **
const char **ps = a; // 错误！
```

27

条款 9

新式转型操作符

在旧式转型（`cast`）下面隐藏着一些见不得人的、鬼鬼祟祟的东西。它们的语法形式使其在一段代码中通常很难引起人们的注意，但它们可能会搞一些可怕的破坏活动，就好比比你冷不丁被一个恶棍猛击一拳似的。让我们阐明旧式转型的含义。显然，在最初的 C 语法中，在表达式中将类型加括号就是旧式转型：

```
char *hopeItWorks = (char *)0x00ff0000; // 旧式转型
```

C++ 引入了另一种转型，即采用函数形式的转型语法来表达同样的意思：

```
typedef char *PChar;  
hopeItWorks =  
    PChar( 0x00ff0000 ); // 函数形式 / 旧式转型
```

函数形式的转型也许看上去比它那可怕的老祖先斯文一些，但实际上同样龌龊，应该像躲避瘟疫一样躲避它们。

诚实可靠的程序员使用新式转型操作符，因为它们能够更精确地表达意思。有四个新式转型操作符，每一个都有着特定的用途。

`const_cast` 操作符允许添加或移除表达式中类型的 `const` 或 `volatile` 修饰符：

```
const Person *getEmployee() { ... }  
//...  
Person *anEmployee = const_cast<Person *>(getEmployee());
```

在以上代码中，使用 `const_cast` 来剥除 `getEmployee` 返回类型中的 `const` 修饰符。可以使用旧式转型获得同样的效果：

```
anEmployee = (Person *)getEmployee();
```

但使用 `const_cast` 的做法更好，这有几方面的原因。首先，它看上去丑陋、醒目，就像一个受伤的拇指一样从代码中伸出来。这是件好事，因为任何形式的转型都存在危险。所以，它们写起来应该很痛苦，只有当不得不使用时才键入它们。它们还应该易于发现，因为无论何时当代码中出现 `bug` 时，人们应该首先检查转型这个嫌犯。其次，

`const_cast` 要比旧式转型的威力小，因为它只影响类型修饰符。这个限制同样是件好事，因为它允许我们更精确地表达意图。使用旧式转型等于告诉编译器，“你给我闭嘴，因为我希望将 `getEmployee` 的返回类型转换为 `Person *`”。而使用 `const_cast` 则等于告诉编译器，“我只是希望将 `getEmployee` 的返回类型的 `const` 去掉”。就目前来看，这两种语句没有大的差别（尽管实际上它们都很无礼），但是对 `getEmployee` 函数进行了一些维护性的修改后，情况就不同了：

```
const Employee *getEmployee(); // 大修改！
```

旧式转型所强加的规则现在仍然有效，编译器将不会对从 `const Employee *` 到 `Person *` 这个不正确的转换作出反应，但如果使用 `const_cast`，编译器将会发出抱怨，因为这么剧烈的转换已经超出了它的能力范围。简而言之，`const_cast` 优于旧式转型，因为它更丑陋、更难用，并且威力较小。

`static_cast` 操作符用于相对而言可跨平台移植的转型。最常见的情况是，它用于将一个继承层次结构中的基类的指针或引用，向下转型为一个派生类的指针或引用（参见“能力查询[条款27]”）：

```
Shape *sp = new Circle;
Circle *cp = static_cast<Circle *>(sp); // 向下转型
```

在这个例子中，使用 `static_cast` 将会产生正确的代码，因为 `sp` 确实指向一个 `Circle` 对象。然而，如果 `sp` 指向其他类型的 `Shape`^①，那么当使用 `cp` 时，很可能会得到某种运行期错误。因此，重申一遍，虽然这种新式转型操作符比旧式转型安全一些，但还不够安全。

30 注意，`static_cast` 无法像 `const_cast` 那样改变类型修饰符。这意味着，有时需要使用由两个新式转型操作符所构成的转型序列，来获得单个旧式转型所能达到的效果：

```
const Shape *getNextShape() { ... }
//...
Circle *cp =
    static_cast<Circle *>(const_cast<Shape *>(getNextShape()));
```

标准并没有对 `reinterpret_cast` 的行为提供太多的保证，不过它通常的行为可以顾名思义。它从位（bit）的角度来看待一个对象，从而允许将一个东西看作另一个完全不同的东西：

① 所谓“其他类型的 `Shape`”，是指 `Shape` 的其他派生类实例。本书中多次出现类似的说法，请读者留意。——译者注

```

hopeItWorks = // 把int假装成指针
    reinterpret_cast<char*>(0x00ff0000);
int *hopeless = // 把char*假装成int*
    reinterpret_cast<int*>(hopeItWorks);

```

这类东西在低层编码里偶尔非用不可，但它可能不具移植性。你要慎重行事。注意区分当分别使用`reinterpret_cast`和`static_cast`将指向基类的指针向下转型为指向派生类的指针时的行为。`reinterpret_cast`通常只是将基类指针假装成一个派生类指针而不改变其值，而`static_cast`（以及旧式转型——从这个角度来说）则将执行正确的地址操作（参见“指针比较的含义[条款28]”）。

在类层次结构的范畴谈转型，就会涉及到`dynamic_cast`。`dynamic_cast`通常用于执行从指向基类的指针安全地向下转型为指向派生类的指针（请参考“能力查询[条款27]”）。不同于`static_cast`的是，`dynamic_cast`仅用于对多态类型进行向下转型（也就是说，被转型的表达式类型，必须是一个指向带有虚函数的类类型的指针），并且执行运行期检查工作，来判定转型的正确性。当然，这种安全性的获得是要付出代价的。使用`static_cast`通常无需付出（或付出极少）运行期代价，而使用`dynamic_cast`则意味着要付出显著的运行期开销。

```

const Circle *cp =
    dynamic_cast<const Circle*>( getNextShape() );
if( cp ) { ... }

```

31

如果`getNextShape`返回一个指向`Circle`的指针（或者从`Circle`公有派生的东西，换句话说，一些和`Circle`之间存在着is-a关系的东西。参见“多态[条款2]”），那么转型就是成功的，并且`cp`将会指向一个`Circle`。否则`cp`将为空。注意，我们可以将声明和测试结合于同一个表达式中：

```

if( const Circle *cp
    = dynamic_cast<const Circle*>(getNextShape()) ) { ... }

```

这样做是有好处的，因为它将变量`cp`的作用域限制在`if`语句之内，因此，当不再使用它时，`cp`将会离开作用域（并被销毁）。

有关`dynamic_cast`一个不太常见的用法是对引用类型执行向下转型：

```

const Circle &rc = dynamic_cast<const Circle &>(*getNextShape());

```

这个操作类似于对指针类型的`dynamic_cast`操作，不过如果转型失败，操作符将抛出一个`std::bad_cast`异常而不是仅仅返回一个空指针（记住，不存在空引用！参见“引

用是别名而非指针[条款5]”)。习惯上, 对一个指针进行 `dynamic_cast` 等于在说: “这个 `Shape` 指针真的指向一个 `Circle` 吗? 如果不是, 我可以处理这种情况”。而对一个引用执行 `dynamic_cast` 则等于声明一个不变式 (invariant): “这个 `Shape` 应该是一个 `Circle`, 否则, 肯定是哪儿出了严重的错误!”。

与其他新式转型操作符相比, `dynamic_cast` 只是偶尔需要使用, 但因为它背上了“安全”的名声, 所以常常被滥用。参见“Factory Method 模式[条款30]”以便了解一个滥用的例子。

条款 10

常量成员函数的含义

从技术上来说，常量成员函数很简单。从社会意义来说，它们却可能很复杂。

在类 `X` 的非常量成员函数中，`this` 指针的类型为 `X *const`。也就是说，它是指向非常量 `X` 的常量指针（参见“常量指针与指向常量的指针[条款 7]”）。由于 `this` 指向的对象不是常量，因此它可以被修改。而在类 `X` 的常量成员函数中，`this` 的类型为 `const X* const`。也就是说，是指向常量 `X` 的常量指针。由于指向的对象是常量，因此它不能被修改。这就是常量成员函数和非常量成员函数之间的区别。

这也是为什么可以使用常量成员函数来改变对象的逻辑状态原因，虽然对象的物理状态没有发生改变。考虑如下类 `X` 的实现，这个实现很普通，`X` 使用一个指向已分配的缓冲区的指针，来保存它的一些状态：

```
class X {
public:
    X() : buffer_(0), isComputed_(false) {}
    //...
    void setBuffer() {
        int *tmp = new int[MAX];
        delete [] buffer_;
        buffer_ = tmp;
    }

    void modifyBuffer( int index, int value ) const // 不道德!
    { buffer_[index] = value; }

    int getValue() const {
        if( !isComputed_ ) {
            computedValue_ = expensiveOperation(); // 错误!
            isComputed_ = true; // 错误!
        }
        return computedValue_;
    }
}
```

```
private:
    static int expensiveOperation();
    int *buffer_;
    bool isComputed_;
    int computedValue_;
};
```

setBuffer 成员函数必须是非常量的，因为它要修改其所属的 X 对象的一个数据成员。然而，modifyBuffer 可以被合法地标为常量，因为它没有修改 X 对象，它只是修改 X 的 buffer_ 成员所指向的一些数据。

这种做法是合法的，但很不道德。就像那些口口声声说尊重法律条文实际上却在违背其本意的奸诈律师一样，一个编写可以改变对象逻辑状态的常量成员函数的 C++ 程序员，即使编译器未宣判他有罪，他的同事也会判他有罪，因为这种做法很不厚道！

话又说回来。有时一个真的应该被声明为常量的成员函数必须要修改其对象。这常见于利用“缓式求值 (lazy evaluation)”机制来计算一个值时。换句话说，只有当第一次提出请求，才计算值，目的在于在该请求根本没有发出的其余情形下，让程序运行更快。函数 X::getValue 试图对一个代价高昂的计算执行“缓式评估”，但是，由于它被声明为常量成员函数，因此不允许它设置对象的 isComputed_ 和 computedValue_ 数据成员的值。在这种情况下会有一个进行转型犯错的诱惑，为的是能够让事情变得更好，即将该成员函数声明为常量：

```
int getValue() const {
    if( !isComputed_ ) {
        X *const aThis = const_cast<X *const>(this); // 糟糕的念头！
        aThis->computedValue_ = expensiveOperation();
        aThis->isComputed_ = true;
    }
    return computedValue_;
}
```

34

千万抵制住这个诱惑！处理这种情形的正确方式是将有关数据成员声明为 mutable：

```
class X {
public:
    //...
    int getValue() const {
        if( !isComputed_ ) {
            computedValue_ = expensiveOperation(); // 很好
            isComputed_ = true; // 也很好
        }
    }
};
```

```

    }
    return computedValue_;
}
private:
    //...
    mutable bool isComputed_; // 现在可以修改了
    mutable int computedValue_; // 现在可以修改了
};

```

类的非静态数据成员可以声明为mutable, 这将允许它们的值可以被该类的常量成员函数（当然也包括非常量成员函数）修改，从而允许一个“逻辑上为常量”的成员函数被声明为常量，虽然其实现需要修改该对象。

对成员函数的 this 指针类型加上常量修饰，就可以解释函数重载解析是如何区分一个成员函数的常量和非常量版本的。下面是一个常见的重载索引操作符的例子：

```

class X {
public:
    //...
    int &operator [] (int index);
    const int &operator [] (int index) const;
    //...
};

```

35

我们可以回想起二元重载成员操作符的左实参是作为 this 指针传入的。因此，当对一个 X 对象进行索引操作时，X 对象的地址被作为 this 指针传入：

```

int i = 12;
X a;
a[7] = i; // this 是 X *const, 因为 a 是非常量
const X b;
i = b[i]; // this 是 const X *const, 因为 b 是常量

```

重载解析会将常量对象的地址和指向常量的 this 指针相匹配。作为另一个例子，考虑如下具有两个常量参数的非成员二元操作符：

```

X operator +( const X &, const X & );

```

如果决定声明一个与此重载操作符对应的成员形式的对应物，应该将其声明为常量成员函数，目的是为了保持的左实参的常量性质：

```
class X {  
    public:  
        //...  
        X operator +( const X &rightArg ); // 左边的参数是非常量!  
        X operator +( const X &rightArg ) const; // 左边的参数是常量  
        //...  
};
```

就像社会生活中的许多领域一样，在 C++ 中正确地使用常量编程在技术上很简单，但在道德上（精神上）具有一定的挑战性。

编译器会在类中放东西

C 程序员习惯于了解所使用的 `struct` 的内部结构和布局的一切知识，并且习惯于编写依赖于特定布局的代码。Java 程序员习惯于在对所使用对象的结构布局懵懂无知的情况下编程，并且往往会认为一旦开始使用 C++ 编程，那种懵懂无知的日子就结束了。实际上，在 C++ 中，安全和可移植的编程实践，确实需要对类对象的结构和布局保持某种程度的“健康的不知情”。

对于一个类而言，并非总是“所见即所得”。举个例子，大多数 C++ 程序员都知道，如果一个类声明了一个或多个虚函数，那么编译器将会为该类的每一个对象插入一个指向虚函数表的指针（事实上，标准并没有保证一定如此，但所有现有的 C++ 编译器都是采用这种方式来实现虚函数机制的）。然而，处于堪可胜任和游刃有余水平之间的 C++ 程序员通常会假定，不同平台之间的虚函数表指针的位置是相同的，因此编写依赖于这种假定的代码，从而导致致命的错误。实际上，有些编译器将指针置于对象的开头，还有一些则将其放在对象的末尾，而且，如果涉及多重继承，若干个虚函数表指针就可能会散布于对象之中。所以，永远不要做这样的假定，永远不要想当然！

且慢，我的话还没说完。如果使用了虚拟继承（virtual inheritance），对象将会通过嵌入的指针、嵌入的偏移或其他非嵌入的信息来保持对其虚基类子对象（virtual base subobject）位置的跟踪。因此，即便类没有声明虚函数，其中还是有可能被插入了一个虚函数表指针！我有没有给你说过这件事：不管类的数据成员的声明顺序如何，编译器都被允许（有节制地）重新安排它们的布局？有什么办法来阻止这种疯狂的行为吗？

37

有！如果确保一个类类型像一个 C `struct` 非常重要，就可以定义一个 **POD**（标准的说法为“**plain old data**（朴素的旧式数据）”）。自然而然，内建的类型，比如 `int`、`double` 以及诸如此类的东西，都是 **POD**，而且 C `struct` 或类似 `union`（`union-like`）的声明，也都是 **POD**。

```
struct S { // 一个 POD struct
    int a;
    double b;
};
```

这样的 POD 可以像对应的 C struct 那样安全地进行操纵（至于说到底有多安全，在 C++ 中就像在 C 中一样值得怀疑）。然而，如果计划对 POD 进行低层的处理，那么，在对代码进行维护的过程中，始终保持其为 POD 很重要，否则所有的赌注将会输得精光：

```
struct S { // 不再是一个 POD struct!
    int a;
    double b;
private:
    std::string c; // 进行了一些维护
};
```

如果不乐意只能处理 POD，那么，编译器的这种多管闲事对应该如何操纵类对象有何暗示呢？暗示就是，应该在高层操纵类对象，而不应该把它当成一组位的集合。在不同的平台上，高层的操作将会做相同的事情，但它们的底层实现可能并不相同。

例如，如果希望复制一个类对象，那么永远都不要使用 memcpy 这样的标准内存块复制函数（或手工编写的等价代码），因为这种函数是用来复制存储区，而不是用来复制对象的（“placement new [条款 35]”讨论了二者之间的区别）。相反，应该使用对象的初始化或赋值操作。对象的构造函数是编译器建立隐藏机制的地方，该隐藏机制实现对象的虚函数以及诸如此类的东西。仅仅往未被初始化的存储区中塞入一大把比特的做法往往无法达到预计的效果。同样的道理，将一个对象复制给另外一个对象时，必须小心不要覆盖这些内部类的机制。例如，赋值操作永远不应该改变对象的虚函数表指针的值，它们由构造函数设置，并且在对象整个生命期内保持不变。简单的“比特冲击”可能还会破坏脆弱的内部结构（参见“复制操作符[条款 13]”）。

38

另一个常见的问题是假定一个类的特定成员位于对象中给定的位置。特别地，有一种并非不常见的情况，就是一些聪明过头的代码，要么假定虚函数表指针位于零偏移处（换句话说，是类中的第一个东西），要么假定第一个声明的数据成员位于零偏移处。在半数以上场合下，这两个假定都是不正确的。当然，任何时候两个假定都不可能同时成立：

```
struct T { // 不是一个 POD
    int a_; // a_ 的偏移量未知
    virtual void f(); // vptr 的偏移量未知
};
```

我不想在这个话题上继续絮叨下去，这样的禁忌列表会变得冗长、无趣而乏味。但是，如果你下一次发现对类的内部结构作了低级的假定，请暂停并反省一下，并且让你的思想远离这种“低级”趣味。

39

条款 12

赋值和初始化并不相同

初始化和赋值是不同的操作，它们具有不同的用途和实现。

直截了当地说，赋值发生于当你赋值时，除此之外，遇到所有其他的复制情形均为初始化，包括声明、函数返回、参数传递以及捕获异常中的初始化。

赋值和初始化本质上是不同的操作，不仅仅因为它们用于不同的上下文，而且还因为它们做的事情不同。对于 `int` 或 `double` 这样的内建类型来说，这种操作上的不同并不明显，因为在这种情况下，赋值和操作不过是简单地复制一些位而已（另请参考“引用是别名而非指针[条款 5]”）：

```
int a = 12; // 初始化，将 0x000C 复制给 a
a = 12; // 赋值，将 0x000C 复制给 a
```

然而，对于用户自定义类型来说，情况却截然不同。考虑如下简单的非标准字符串类：

```
class String {
public:
    String( const char *init ); // 故意不标为 explicit!
    ~String();
    String( const String &that );
    String &operator =( const String &that );
    String &operator =( const char *str );
    void swap( String &that );
    friend const String // 用于连接字符串
        operator +( const String &, const String & );
    friend bool operator <( const String &, const String & );
    //...
private:
    String( const char *, const char * ); // 计算性的构造函数
    char *s_;
};
```

采用字符串初始化一个 `String` 对象很简单。先分配一个足够大的缓冲区，用于容纳该

字符串的复制，然后执行复制动作：

```
String::String( const char *init ) {
    if( !init ) init = "";
    s_ = new char[ strlen(init)+1 ];
    strcpy( s_, init );
}
```

析构函数也很直观：

```
String::~String() { delete [] s_; }
```

赋值比构造复杂一些：

```
String &String::operator =( const char *str ) {
    if( !str ) str = "";
    char *tmp = strcpy( new char[ strlen(str)+1 ], str );
    delete [] s_;
    s_ = tmp;
    return *this;
}
```

赋值有点像一个析构动作后跟一个构造动作。对于复杂的用户自定义类型来说，目标（左侧，或者说 `this`）在采用源（右侧，或者说 `str`）重新初始化之前必须被清理掉。对于 `String` 类型来说，`String` 现有的字符缓冲区在被附加上一个新的字符缓冲区之前必须被释放掉。参见“异常安全的函数[条款39]”以便了解对语句顺序的解释。（顺便提及，就像每周都有人“发明”新思想一样，那种采用显式析构函数调用和使用 `placement new` 调用构造函数来实现赋值的做法并非总能行得通，而且不是异常安全的。不要那么做！）

由于一个正当的赋值操作会清掉左边的实参，因此永远都不应该对一个未初始化的存储区执行用户自定义赋值操作：

```
String *names = static_cast<String *> (::operator new( BUFSIZ ));
names[0] = "Sakamoto"; // 哎呀！delete [] 未被初始化的指针 names！
```

42

在这个例子中，`names` 指向未初始化的存储区，因为我们直接调用了 `operator new`，从而避免了通过 `String` 的默认构造函数执行的隐式初始化动作，因此 `names` 指向一块充满着随机位的内存。当 `String` 赋值操作符在第二行代码中被调用时，它试图对一个未初始化的指针执行一个 `array delete` 操作（参见“`placement new` [条款35]”），以便了解一种执行类似于这种赋值操作的安全方式）。

由于构造函数比赋值操作符做的事少（因为构造函数可以假定它肯定是在处理一个未初始化的存储区），因此，一个实现有时利用所谓的“计算性构造函数（`computational`”

constructor)”来提高效率:

```
const String operator +( const String &a, const String &b )  
    { return String( a.s_, b.s_ ); }
```

这个带有两个参数的计算性构造函数无意成为 String 类接口的一部分, 因此它声明为 private。

```
String::String( const char *a, const char *b ) {  
    s_ = new char( strlen(a)+strlen(b)+1 );  
    strcat( strcpy( s_, a ), b );  
}
```

条款 13

复制操作

复制构造 (copy construction) 和复制赋值 (copy assignment) 是两种不同的操作。从技术角度来说, 它们之间没有丝毫的关联, 但从“社会”角度来说, 它们一般被放到一起, 同时出现, 并且必须兼容:

```
class impl;
class Handle {
public:
    //...
    Handle( const Handle & ); // 复制构造函数
    Handle &operator =( const Handle & ); // 复制赋值操作符
    void swap( Handle & );
    //...
private:
    impl *impl_; // 指向 Handle 的实现
};
```

复制操作的影响是如此深远, 它们甚至需要异乎寻常地遵守惯例。这两个操作总是被成对地声明, 具有如上所示的签名 (另请参考“auto_ptr 非同寻常[条款 43]”以及“禁止复制[32]”)。也就是说, 对于一个类 X 而言, 复制构造函数应该被声明为 X(const X &), 而复制赋值操作符则应该被声明为 X &operator =(const X &)。通常来说, 和传统的非成员形式的 swap 相比, 如果成员形式的 swap 实现具有性能或异常安全的优势, 那么定义一个成员函数 swap 往往是个好主意。典型的非成员形式的 swap 实现是很直观的:

```
template <typename T>
void swap( T &a, T &b ) {
    T temp(a); // 调用 T 的复制构造函数
    a = b; // 调用 T 的复制赋值操作符
    b = temp; // 调用 T 的复制赋值操作符
}
```

45

这个 swap (与标准库中的 swap 一致) 根据类型 T 的复制操作进行定义, 如果 T 的实现短小简单, 这种方式就会工作得很好; 但如果 T 是一个庞大而复杂的类, 这种方式就会导致不小的开销。对于 Handle 这样的类, 我们有更好的方式, 即, 交换指向各自实现

的指针。

```
inline void Handle::swap( Handle &that )
{ std::swap( impl_, that.impl_ ); }
```

下面展示如何编写一个异常安全的复制赋值操作。首先，要得到一个异常安全的复制构造函数和一个异常安全的 swap 操作。剩下的事情就好办了：

```
Handle &Handle::operator =( const Handle &that ) {
    Handle temp( that ); // 异常安全的复制构造
    swap( temp ); // 异常安全的 swap
    return *this; // 我们假定 temp 的析构不会抛出异常
}
```

对于句柄类（**handle class**）来说，这项技术工作得尤其好。句柄类是这样的一种类，它主要或全部是由一个指向其实现的指针构成。如前例所示，为句柄类编写异常安全的 swap 乃是小菜一碟，且效率极高。

对于复制赋值的这个实现来说，微妙之处在于复制构造的行为必须和复制赋值的行为“兼容”。它们尽管是不同的操作，然而此处存在一个影响深远的惯用假定，就是它们产生的结果不应该有区别。也就是说，不管是写成

```
Handle a = ...
Handle b;
b = a; // 将 a 赋给 b
```

还是写成

```
Handle a = ...
Handle b( a ); // 用 a 来初始化 b
```

b 的结果值和将来的行为都应该没有差别，不管它是通过赋值还是通过初始化而得到那个值的。

46

当使用标准容器时，这种兼容性尤其重要，因为它们的实现常常用复制构造来代替复制赋值，当然也就期望两种操作产生一致的结果（参见“placement new [条款 35]”）。

一个或许更常见的复制赋值实现具有如下的结构：

```
Handle &Handle::operator =( const Handle &that ) {
    if( this != &that ) {
        // 进行赋值……
    }
    return *this;
}
```

这种对自身赋值所执行的检查往往是为了正确性（有时也是出于效率方面的考虑）。更确切地说，为了确保赋值表达式的左操作数（`this`）和右操作数（例如 `that`）具有不同的地址。

大多数 C++ 程序员在职业生涯里，都摆弄过实现虚拟复制赋值（`virtual copy assignment`）的思想。这种做法合法但过于复杂，所以别那么做。应该采用 `clone`（克隆）

[47] 取而代之（参见“虚构造函数与 Prototype 模式[条款 29]”）。

条款 14

函数指针

可以声明一个指向特定类型函数的指针：

```
void (*fp)(int); // 指向函数的指针
```

注意，其中的括号是必不可少的，它表明 fp 是一个指向返回值为 void 的函数的指针，而不是返回值为 void* 的函数（参见“处理函数和数组声明[条款 17]”）。就像指向数据的指针一样，指向函数的指针也可以为空，否则它就应该指向一个具有适当类型的函数。

```
extern int f( int );
extern void g( long );
extern void h( int );
//...
fp = f; // 错误! &f 的类型为 int(*) (int) 而非 void(*) (int)
fp = g; // 错误! &g 的类型为 void(*) (long) 而非 void(*) (int)
fp = 0; // OK, 设置为 null
fp = h; // OK, 指向 h
fp = &h; // OK, 明确地赋予函数地址
```

注意，将一个函数的地址初始化或赋值给一个指向函数的指针时，无需显式地取得函数地址，编译器知道隐式地获得函数的地址，因此在这种情况下 & 操作符是可选的，通常省略不用。

类似地，为了调用函数指针所指向的函数而对指针进行解引用操作也是不必要的，因为编译器可以帮你解引用：

```
(*fp)(12); // 显式地解引用
fp(12); // 隐式地解引用，结果相同
```

和 void* 指针可以指向任何类型的数据不同，不存在可以指向任何类型函数的通用指针。还要注意，非静态成员函数的地址不是一个指针，因此不可以将一个函数指针指向一个非静态成员函数（参见“指向成员函数的指针并非指针[条款 16]”）。

函数指针的一个传统用途是实现回调（callback）（另请参考“函数对象[条款 18]”和

“Command 模式与好莱坞法则[条款 19]”，以便了解更有效的回调技术)。一个回调就是一个可能的动作，这个动作在初始化阶段设置，以便在对将来可能发生的事件做出反应时而被调用。打个比方，如果我们希望救火，那么最好事先计划好该如何做出反应：

```
extern void stopDropRoll();
inline void jumpIn() { ... }
//...
void (*fireAction)() = 0;
//...
if( !fatalist ) { // 如果你关心失火了……
    // 那么设置适当的动作，以防万一！
    if( nearWater )
        fireAction = jumpIn;
    else
        fireAction = stopDropRoll;
}
```

一旦决定了要执行的动作，代码中的另一个部分就可以专注于是否以及何时去执行该动作，而无需关心这个动作到底是什么：

```
if( ftemp >= 451 ) { // 如果着火了
    if( fireAction ) // 并且要执行一个动作
        fireAction(); // 执行之！
}
```

注意，一个函数指针指向内联函数（inline function）是合法的。然而，通过函数指针调用内联函数将不会导致内联式的函数调用，因为编译器通常无法在编译期精确地确定将会调用什么函数。在前例中，fireAction可能指向两个函数中的任一个（当然，也可能两个都不指向），因此在调用点，编译器别无他法，只好生成间接、非内联的函数调用代码。

另外，函数指针持有一个重载函数的地址也是合法的：

```
void jumpIn();
void jumpIn( bool canSwim );
//...
fireAction = jumpIn;
```

指针的类型被用于在各种不同的候选函数中挑选最佳匹配的函数。在这个例子中，fireAction的类型为void(*)()，因此选择的是第一个jumpIn函数。

在标准库中，有好几个地方使用了函数指针作为回调机制，最突出的就是被标准函数

`set_new_handler`用于设置回调。当全局`operator new`函数无法履行一个内存分配请求时，该回调函数即被调用。例如：

```
void begForgiveness() {
    logError( "Sorry!" );
    throw std::bad_alloc();
}
//...
std::new_handler oldHandler =
    std::set_new_handler(begForgiveness);
```

标准类型名称`new_handler`是一个typedef：

```
typedef void (*new_handler)();
```

因此，回调函数必须是一个不带参数且返回`void`的函数。`set_new_handler`函数将回调设置为参数，并且返回前一个回调。不存在什么单独的用于获得和设置回调的函数。获得当前回调需采用一种回旋式的惯用手法：

```
std::new_handler current
    = std::set_new_handler( 0 ); // 获取
std::set_new_handler( current ); // 恢复！
```

另外，标准函数`set_terminate`和`set_unexpected`也使用了这种合二为一的get/set回调惯用法。

条款 15

指向类成员的指针并非指针

“指向类成员的指针”这个描述中有“指针”这个术语，其实，这是不合适的，因为它们既不包含地址，行为也不像指针。

如果你已经熟稔常规指针的声明语法，那么声明一个指向成员的指针，语法并不是太可怕：

```
int *ip; // 一个指向 int 的指针
int C::*pimC; // 一个指针，指向 C 的一个 int 成员
```

所要做的全部事情就是使用 `classname::*` 而不是普通的 `*`，来表明你在指向 `classname` 的一个成员。在其他方面，语法与常规的指针声明符一样。

```
void * * *const* weird1;
void *A::*B::*const* weird2;
```

其中 `weird1` 的类型为：一个指针，指向一个常量指针，后者又指向一个指针，后者又指向一个指针，后者则指向 `void`。而 `weird2` 的类型为：一个指针，指向一个常量指针，后者指向 `B` 的一个成员，后者指向一个指针，后者指向 `A` 的一个成员，该成员为一个指向 `void` 的指针（没有别的意思，只是举个例子，在现实编程中不会看到这么复杂、这么变态的声明）。

一个常规的指针包含一个地址。如果解引用该指针，就会得到位于该地址的对象：

```
int a = 12;
ip = &a;
*ip = 0;
a = *ip;
```

与常规指针不同，一个指向成员的指针并不指向一个具体的内存位置，它指向的是一个类的特定成员，而不是指向一个特定对象里的特定成员。通常最清晰的做法，是将指向数据成员的指针看作为一个偏移量。当然，事情未必一定如此，因为 C++ 标准对于一个指向数据成员的指针究竟该如何实现只字未提。标准只是说明了它的语法形式以及必须表现出来的行为。然而，大多数编译器都将指向数据成员的指针实现为一个整数，其中包含被

指向的成员的偏移量，另外加上 1（加 1 是为了让值 0 可以表示一个空的数据成员指针）。这个偏移量告诉你，一个特定成员的位置距离对象的起点有多少个字节。

```
class C {
public:
    //...
    int a_;
};

int C::*pimC; // 一个指针，指向 C 的一个 int 成员
C aC;
C *pC = &aC;
pimC = &C::a_;
aC.*pimC = 0;
int b = pC->*pimC;
```

将 `pimC` 的值设置为 `&C::a_` 时，实际上是将 `pimC` 设置为 `a_` 在 `C` 内的偏移量。说得明白一些，除非 `a_` 是静态成员，否则在表达式 `&C::a_` 中使用 `&` 并不会带来一个地址，而是一个偏移量。注意，这个偏移量适用于类型 `C` 的任何对象，换句话说，如果在一个 `C` 对象内成员 `a_` 距离起点的偏移为 12 字节，那么在任何其他 `C` 对象中，`a_` 距离起点的偏移都是 12 字节。

给定一个成员在类内的偏移量，为了访问位于那个偏移量的数据成员，我们需要该类的一个对象的地址。这时候就需要 `*` 和 `->*` 这两个看上去非同寻常的操作符闪亮登场了。当写下 `pC->*pimC` 时，其实是请求将 `pC` 内的地址加上 `pimC` 内的偏移量，为的是访问 `pC` 所指向的 `C` 对象中适当的数据成员。当写 `aC.*pimC` 时，是在请求 `aC` 的地址加上 `pimC` 中的偏移量，也是为了访问 `pC` 所指向的 `C` 对象中适当的数据成员。

指向数据成员的指针不如指向成员函数的指针那么常用，但它们对于描述“逆变性 (contravariance)”的概念很方便。在 C++ 中，存在从指向派生类的指针到指向其任何公有基类的预定义转换。我们常说在派生类与其公共基类之间存在着 is-a 关系，并且这种关系在对问题领域进行分析时会自然而然地出现（参见“多态[条款 2]”）。例如，可以宣称一个 `Circle` 是一个 `Shape`（通过公有继承），而且 C++ 通过提供从 `Circle*` 到 `Shape*` 的隐式转换来支持这个宣称。注意，不存在从 `Shape*` 到 `Circle*` 的隐式转换，这样的转换毫无意义，因为可能存在许多不同类型的 `Shape`，它们并不全是 `Circle`。（说“一个 `Shape` 是一个 `Circle`”，听起来很弱智，不是吗？）

在指向类成员的指针的情况下则恰恰相反：存在从指向基类成员的指针到指向公有派生类成员的指针的隐式转换，但不存在从指向派生类成员的指针到指向其任何一个基类成员的

指针的转换。这个逆变性概念看来有违直觉，不过，如果回忆起指向数据成员的指针并非指向一个对象的指针，而只是对象内的一个偏移，就会明白了。

```
class Shape {  
    //...  
    Point center_;  
    //...  
};  
class Circle : public Shape {  
    //...  
    double radius_;  
    //...  
};
```

因为一个Circle也是一个Shape，所以一个Circle对象内包含一个Shape子对象。因而，Shape内的任何偏移量在Circle内也是一个有效的偏移量。

```
Point Circle::*loc = &Shape::center_; // OK, 从基类到派生类的转换
```

然而，一个Shape未必是一个Circle，因此一个Circle的成员的偏移量在Shape内未必是一个有效的偏移量。

```
double Shape::*extent =  
    &Circle::radius_; // 错误！从派生类到基类的转换
```

55

说一个Circle里包含其基类Shape中的所有数据成员，这是有意义的（也就是说，它从Shape那里继承了那些成员），C++支持我们的说法，它提供了从指向Shape的成员的指针到指向Circle的成员的指针的隐式转换。说一个Shape包含Circle中所有的数据成员，是没有意义的（Shape没有从Circle那里“继承”任何东西），C++不允许从指向Circle成员的指针转换到指向Shape成员的指针，以此来提醒我们这一点。

条款 16

指向成员函数的指针并非指针

获取非静态成员函数的地址时，得到的不是一个地址，而是一个指向成员函数的指针。

```
class Shape {
public:
    //...
    void moveTo( Point newLocation );
    bool validate() const;
    virtual bool draw() const = 0;
    //...
};

class Circle : public Shape {
    //...
    bool draw() const;
    //...
};

//...
void (Shape::*mf1)( Point ) = &Shape::moveTo; // 不是指针
```

指向成员函数的指针的声明语法一点都不比指向常规函数的指针的语法来得困难（不可否认，从目前的情况来看，指向常规函数的指针的语法已经够糟糕的了。参见“处理函数和数组声明符[条款 17]”）。与指向数据成员的指针一样，我们需要做的就是使用 `classname::*` 而不是 `*` 来指明所指向的函数是 `classname` 的一个成员。然而，和指向常规函数的指针不同，指向成员函数的指针可以指向一个常量成员函数：

```
bool (Shape::*mf2)() const = &Shape::validate;
```

和指向数据成员的指针的情形一样，为了对一个指向成员函数的指针进行解引用，需要一个对象或一个指向对象的指针。对于指向数据成员的指针的情形，为了访问该成员，需要将对象的地址和成员的偏移量（包含于指向数据成员的指针中）相加。对于指向成员函数的指针的情形，需要将对象的地址用作（或用于计算，参见“指针比较的含义[条款 28]”）`this` 指针的值，进行函数调用，以及作为其他用途。

```
Circle circ;
Shape *pShape = &circ;
(pShape->*mf2)(); // 调用 Shape::validate
(circ.*mf2)(); // 调用 Shape::validate
```

->* 和 .* 操作符必须加上圆括号，因为它们比()操作符优先级低，而且在调用函数之前首先应该找到这个函数。这与在 (a+b)*c 之类的表达式中的括号用法完全类似。在该表达式中，我们希望确保较低优先级的加法在较高优先级的乘法之前执行。

注意，不存在什么指向成员函数的“虚拟”指针。虚拟性是成员函数自身的属性，而不是指向它的指针所具有的属性。

```
mf2 = &Shape::draw; // draw是虚函数
(pShape->*mf2)(); // 调用 Circle::draw
```

这就是为何一个指向成员函数的指针，通常不能被实现为一个简单的指向函数的指针。一个指向成员函数的指针的实现自身必须存储一些信息，诸如它所指向的成员函数是虚拟的还是非虚拟的，到哪里去找到适当的虚函数表指针（参见“编译器会在类中放东西[条款11]”），从函数的 this 指针加上或减去的一个偏移量（参见“指针比较的含义[条款28]”），以及可能还有其他一些信息。指向成员函数的指针通常实现为一个小型结构，其中包含这些信息。当然，也可以使用其他一些实现。解引用和调用一个指向成员函数的指针通常涉及到检查这些存储的信息，并有条件地执行适当的虚拟或非虚拟的函数调用序列。

和指向数据成员的指针一样，指向成员函数的指针也表现出一种逆变性，即存在从指向基类成员函数的指针到指向派生类成员函数指针的预定义转换，反之则不然。这很好理解，可以考虑基类成员函数会试图通过其 this 指针只访问基类成员，然而派生类函数可能会试图访问基类中不存在的成员。

```
class B {
public:
    void bset( int val ) { bval_ = val; }
private:
    int bval_;
};
class D : public B {
public:
    void dset( int val ) { dval_ = val; }
private:
    int dval_;
```

```
};  
B b;  
D d;  
void (B::*f1)(int) = &D::dset; // 错误! 不存在这种反向转换  
(b.*f1)(12); // 哎呀! 访问不存在的 dval 成员!  
void (D::*f2)(int) = &B::bset; // OK, 存在这种转换  
(d.*f2)(11); // OK, 设置继承来的 bval_ 数据成员
```

条款 17

处理函数和数组声明

指向函数的指针声明与指向数组的指针声明容易混淆，原因在于函数和数组修饰符的优先级比指针修饰符的优先级高，因此通常需要使用圆括号。

```
int *f1(); // 一个返回值为 int * 的函数
int (*fp1)(); // 一个指针，指向一个返回值为 int 的函数
```

具有高优先级的数组修饰符存在同样的问题：

```
const int N = 12;
int *a1[N]; // 一个具有 N 个 int * 元素的数组
int (*ap1)[N]; // 一个指针，指向一个具有 N 个 int 元素的数组
```

理所当然，一旦拥有指向函数或数组的指针，就可以拥有指向这种指针的指针：

```
int (**ap2)[N]; // 一个指针，它指向一个指针，后者则指向一个具有 N 个 int 元素的数组
int *(*ap3)[N]; // 一个指针，指向一个具有 N 个 int* 元素的数组
int (**const fp2)() = 0; // 一个常量指针，指向一个指向函数的指针
int *(*fp3)(); // 一个指针，指向一个返回值为 int * 的函数
```

注意，参数和返回值都会影响函数或函数指针的类型。

```
char *(*fp4)(int,int);
char *(*fp5)(short,short) = 0;
fp4 = fp5; // 错误！类型不匹配
```

当函数和数组修饰符出现于同一个声明中时，事情的复杂性会变得难以估量。考虑如下常见但错误的声明，它试图声明一个函数指针数组：

```
int (*)()afp1[N]; // 语法错误！
```

在以上错误的声明中，函数修饰符()的出现表示到了声明的末尾，而后面附加的 afp1 则表示一个语法错误的开始。这类似于以下的数组声明写法：

```
int[N] a2; // 语法错误！
```

这在 Java 中是合法的，但在 C++ 中是非法的。函数指针数组的正确声明方式，是将数组名字和简单的函数指针声明放在一起。因此可以声明一个装有这些东西的数组：

```
int (*afp2[N])(); // 一个具有N个元素的数组，其元素类型为指向“返回值为int”的函数的指针
```

至此，事情开始变得笨拙，typedef 闪亮登场的时机到了：

```
typedef int (*FP)(); // 一个指向返回值为int的函数的指针
FP afp3[N]; // 一个具有N个“类型为FP”的元素的数组，该类型与afp2相同
```

使用typedef可以简化复杂的声明语法，这也是对你的代码维护者的关爱。使用typedef，甚至标准的set_new_handler函数的声明都变得简单多了：

```
typedef void (*new_handler)();
new_handler set_new_handler( new_handler );
```

如此一来，new_handler（参见“函数指针[条款14]”）是指向这种函数的指针：它不带任何参数，返回void。而set_new_handler则是一个函数，带有一个new_handler作为参数，并返回一个new_handler作为结果。简单吧？如果尝试不用typedef，你的声望在那些维护你的代码的人中将急剧下跌：

```
void (*set_new_handler(void (*)(void)))(void); // 语法没错，但邪恶！
```

还可以声明函数引用：

```
int aFunc( double ); // 函数
int (&rFunc)(double) = aFunc; // 函数引用
```

函数引用很少使用，其应用程度跟常量函数指针差不多：

```
int (*const pFunc)(double) = aFunc; // 常量函数指针
```

数组引用确实提供了一些数组指针所未提供的额外能力，参见“数组形参[条款6]”中的讨论。

条款 18

函数对象

有时需要一些行为类似于函数指针的东西，但函数指针显得笨拙、危险而且过时（让我们承认这一点）。通常最佳方式是使用函数对象（function object）取代函数指针。

与智能指针（参见“智能指针[条款 42]”）一样，函数对象也是一个普通的类对象。智能指针类型重载 `->` 和 `*`（可能还有 `->*`）操作符，来模仿指针的行为；而函数对象类型则重载函数调用操作符 `()`，来创建类似于函数指针的东西。考虑如下函数对象，它的每次调用都计算众所周知的斐波纳契数列（1、1、2、3、5、8、13，...）的下一个元素值：

```
class Fib {
public:
    Fib() : a0_(1), a1_(1) {}
    int operator ()();
private:
    int a0_, a1_;
};

int Fib::operator () {
    int temp = a0_;
    a0_ = a1_;
    a1_ = temp + a0_;
    return temp;
}
```

函数对象就是常规的类对象，但是可以采用标准的函数调用语法来调用它的 `operator()` 成员（此成员可能具有多个重载版本）。

```
Fib fib;
//...
cout << "next two in series: " << fib()
      << ' ' << fib() << endl;
```

63

`fib()` 语法被编译器识别为对 `fib` 对象的 `operator()` 成员函数的调用，这在意思上和 `fib.operator()` 等价，但看起来更简洁。在这个例子中，使用函数对象而不是函数或函数指针的优势在于，用于计算斐波纳契数列下一个值的状态被存储于 `Fib` 对象自身之中。如果

采用函数来实现计算功能，那么必须求助于全局或局部静态变量或其他一些基本的技巧，以便在函数调用之间保持状态，或者将状态信息明确地传递给函数。还要注意的，有别于使用静态数据的函数，我们可以拥有多个同时计算的Fib对象，且其计算过程和结果不会互相干扰。

```
int fibonacci () {
    static int a0 = 0, a1 = 1; // 有问题……
    int temp = a0;
    a0 = a1;
    a1 = temp + a0;
    return temp;
}
```

还有可能并且常见的是获得虚函数指针的效果，这是通过创建一个带有虚拟operator()的函数对象层次结构而实现的。考虑一个数值积分软件，它用于计算曲线所包围面积的近似值，如图5所示。

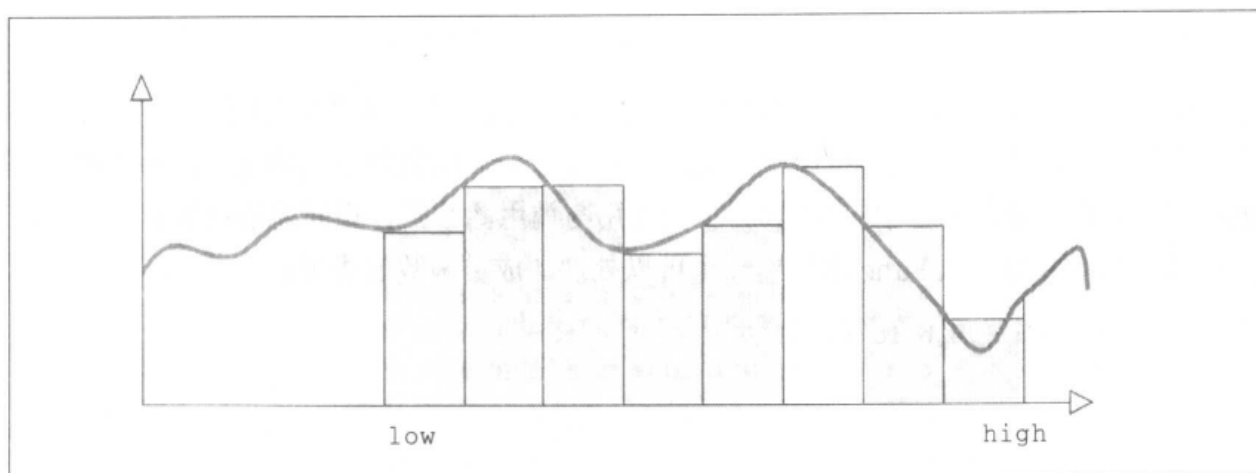


图5 通过对矩形面积求和来进行数值积分（此为简化方法）

一个积分函数可以对low和high之间的值反复调用一个函数，来近似计算曲线所包围的面积，这是通过计算矩形面积的总和而实现的（当然也可以采用一些类似的机制）：

```
typedef double (*F)( double );
double integrate( F f, double low, double high ) {
    const int numsteps = 8;
    double step = (high-low)/numSteps;
    double area = 0.0;
    while( low < high ) {
        area += f( low ) * step;
        low += step;
    }
}
```

```
    return area;
}
```

在这个版本中，传递一个函数指针来执行所期望的积分操作。

```
double aFunc( double x ) { ... }
//...
double area = integrate( aFunc, 0.0, 2.71828 );
```

这可行，但不灵活，因为它使用一个函数指针来指示待整合的函数。它不能处理需要状态的函数或指向成员函数的指针。一个替代的方式是创建一个函数对象层次结构。该层次结构的基类是一个简单接口类，只声明了一个纯虚 `operator()`。

```
class Func {
public:
    virtual ~Func();
    virtual double operator()( double ) = 0;
};
double integrate( Func &f, double low, double high );
```

现在 `integrate` 能够与任何类型的 `Func` 函数对象^①（参见“多态[条款2]”）进行整合。还有一个值得注意的有趣的地方，就是 `integrate` 函数体不需要进行任何修改（当然，重新编译一遍是免不了的），因为我们使用与调用函数指针相同的语法来调用一个函数对象。例如，可以从 `Func` 派生出一个可以处理非成员函数的类型：

```
class NMFunc : public Func {
public:
    NMFunc( double (*f)( double ) ) : f_(f) {}
    double operator()( double d ) { return f_( d ); }
private:
    double (*f_)( double );
};
```

这就允许最初版本的 `integrate` 整合所有的函数：

```
double aFunc( double x ) { ... }
//...
NMFunc g( aFunc );
double area = integrate( g, 0.0, 2.71828 );
```

① 所谓“任何类型的 `Func` 函数对象”，是指任何类型的 `Func` 派生类实例。——译者注

通过为指向成员函数的指针和类对象包装一个适当的接口, 还可以将成员函数整合进来 (参见“指向成员函数的指针并非指针[条款 16]”):

```
template <class C>
class MFunc : public Func {
public:
    MFunc( C &obj, double (C::*f)(double) )
        : obj_(obj), f_(f) {}
    double operator ()( double d )
        { return (obj_.*f_)( d ); }
private:
    C &obj_;
    double (C::*f_)( double );
};
//...
AClass anObj;
MFunc<AClass> f( anObj, &AClass::aFunc );
double area = integrate( f, 0.0, 2.71828 );
```

Command 模式与好莱坞法则

当一个函数对象用作回调时，就是一个 Command（命令）模式的实例。

什么是回调？假设你计划开始一次长途旅行，为此我借给你一辆汽车。鉴于汽车的情况，我可能同时还交给你一个密封的信封，里面装有一个电话号码，并嘱咐你，如果发现发动机出了故障，你可以拨打信封里的电话号码。这就是回调。你事先不必知道电话号码（它可能是一个口碑良好的汽车修理铺电话，也可能是一个公共汽车运输公司的电话，甚至可能是城市垃圾场的电话），实际上，你可能不需要拨打那个电话。从效果上说，处理“发动机故障”事件的任务，已经被你（也就是大家知道的“框架”）和我（也就是大家知道的“框架的客户”）一分为二。作为框架的你，知道何时该去干一些事情，但具体干什么，你一无所知。而我则知道当发生一个特定的事件时，应该干些什么，但不知道何时去干这件事情。我们共同构成一个完整的应用程序。

回调是一种常见的编程技术，传统上被实现为一个指向函数的简单指针（参见“函数指针[条款 14]”）。例如，考虑一个交互式按钮类型，它在屏幕上显示一个带标签的按钮，并且当被点击时执行一个动作。

```
class Button {
public:
    Button( const string &label )
        : label_(label), action_(0) {}
    void setAction( void (*newAction)() )
        { action_ = newAction; }
    void onClick() const
        { if( action_ ) action_(); }
private:
    string label_;
    void (*action_)();
    //...
};
```

Button的用户设置回调函数，然后将Button移交给框架代码，后者可以侦测Button何时被点击了，并执行指定的动作。

```
extern void playMusic();
//...
Button *b = new Button( "Anoko no namaewa" );
b->setAction( playMusic );
registerButtonWithFramework( b );
```

这种责任的分割通常被称为“好莱坞法则”，即“不要 call 我们，我们会 call 你”^①。将按钮设置为执行正确的动作（如果它被点击了），而框架代码则知道，如果按钮被点击了就去调用该动作。

然而，使用一个简单的函数指针作为回调的做法具有一些严格的限制。我们知道，函数往往需要一些数据才能工作，但一个函数指针没有相关联的数据。在上面的例子中，函数 playMusic 如何知道播放什么歌曲？快速修复这个问题的常见做法有二。其一，将函数的作用域大幅缩小，如下：

```
extern void playAnokoNoNamaewa();
//...
b->setAction( playAnokoNoNamaewa );
```

其二，借助于一些声名狼藉的、危险的编程实践，例如使用一个全局变量：

```
extern const MP3 *theCurrentSong = 0;
//...
const MP3 anokoNoNamaewa ( "AnokoNoNamaewa.mp3" );
theCurrentSong = &anokoNoNamaewa;
b->setAction( playMusic );
```

一种具有代表性的更好方式是使用函数对象代替函数指针。将一个函数对象（典型情况为函数对象层次结构）与“好莱坞法则”相结合使用，即为 Command 模式的一个实例。

68

使用这种面向对象方式显而易见的好处是，函数对象可以封装数据。另一个好处是函数对象可以通过虚拟成员表现出动态行为。换句话说，可以拥有一个相关的函数对象的层次结构（参见“函数对象[条款18]”）。此外还获得了第三个好处，稍后再谈。首先使用 Command 模式重新设计 Button 类：

```
class Action { // Command
public:
    virtual ~Action();
    virtual void operator ()() = 0;
    virtual Action *clone() const = 0; // 原型 (Prototype)
```

^① Don't call us, we'll call you, call 一语双关，原意是“打电话”，此处为“调用”。好莱坞法则实际上就是模式中常用的控制转置原则。——译者注

```
};
class Button {
public:
    Button( const std::string &label )
        : label_(label), action_(0) {}
    void setAction( const Action *newAction ) {
        Action *temp = newAction->clone();
        delete action_;
        action_ = temp;
    }
    void onClick() const
        { if( action_ ) (*action_)(); }
private:
    std::string label_;
    Action *action_; // Command
    //...
};
```

现在，Button可以和任何“是一个(is-a)”Action的函数对象协作，比如下面这个：

```
class PlayMusic : public Action {
public:
    PlayMusic( const string &songFile )
        : song_(song) {}
    void operator ()(); // 播放歌曲
private:
    MP3 song_;
};
```

被封装的数据（此例中为待播放的歌曲）既保持了PlayMusic函数对象的灵活性，也保持了它的安全性。

```
Button *b = new Button( "Anoko no namaewa" );
auto_ptr<PlayMusic>
    song( new PlayMusic( "AnokoNoNamaewa.mp3" ) );
b->setAction( song );
```

那么，先前提到的Command模式的第三个神秘的好处是什么呢？简单地说，就是可以处理类层次结构而不是较为原始的、缺乏灵活性的结构（如函数指针）。有了Command层次结构，就可以将Prototype模式和Command模式组合，从而产生可克隆的命令（clonable command）（参见“虚构造函数与Prototype模式[条款29]”）。按照这种套路，我们可以继续将其他模式与Command模式、Prototype模式组合使用，从而获得更大的灵活性。

条款 20

STL 函数对象

如果没有 STL，我们的日子该怎么过？STL 不但使我们能够更轻松、更快捷地编写复杂的代码，而且编写的代码既标准又高度优化。

```
std::vector<std::string> names;
//...
std::sort( names.begin(), names.end() );
```

STL 另一个优雅之处在于高度可配置。在以上的代码中，使用 string 的小于 (<) 操作符对 vector 中的 string 元素进行排序，但在其他场合下，未必总有一个小于操作符可供使用，而且有时并不希望以升序方式进行排序。

```
class State {
public:
    //...
    int population() const;
    float aveTempF() const;
    //...
};
```

类 State 用于表示联邦的一个州，它没有小于操作符，而且也不打算为它实现一个，因为“一个州小于另一个州”说不清是什么意思。（比较名字、人口、被起诉官员的百分比吗？抑或其他？）幸运的是，对于这样的情形来说，STL 一般允许我们指定一个替代的类似小于操作符（less-than-like）的操作。这样的操作被称为“比较器”，因为它用于比较两个值：

```
inline bool popLess( const State &a, const State &b )
{ return a.population() < b.population(); }
```

71

拥有针对 State 的比较器之后，就可以用它进行排序了：

```
State union[50];
//...
std::sort( union, union+50, popLess ); // 按人口进行排序
```

这里我们传递一个指向 popLess 函数的指针作为比较器（请回忆一下，当一个函数名字作为参数进行传递时，函数名字会退化成一个指针，正如此处数组名字 union 退化成一个指向其首元素的指针那样）。因为 popLess 作为函数指针进行传递，所以它在 sort 内无法被内联。如果希望得到快速的排序操作，这种做法只能让人感到遗憾了（参见“函数指针[条款 14]”）。

如果使用函数对象作为比较器，情况就会好得多：

```
struct PopLess : public std::binary_function<State,State,bool> {
    bool operator ()( const State &a, const State &b ) const
    { return popLess( a, b ); }
};
```

PopLess 类型是一个典型的、有着正确构造的 STL 函数对象的例子。首先，它是一个函数对象。它重载了函数调用操作符，因此可以以普通函数调用的语法调用。这一点很重要，因为诸如 sort 这样的 STL 泛型算法是以这种方式编写的：函数指针和函数对象都可以用来实例化它们，只要此二者可以采用典型的函数调用语法进行调用即可。一个具有重载的 operator() 的函数对象完全满足这个语法要求。

其次，它派生于标准的 binary_function 基类。此项机制允许其他部分的 STL 实现询问函数对象编译期问题（参见“嵌入的类型信息[条款 53]”）。在这个例子中，从 binary_function 派生下来的 PopLess 类型允许我们找出函数对象的参数和返回值类型。不过在这里我们并没有利用这种能力，但是可以打赌肯定有人需要这样的能力，而且希望我们的 PopLess 类型可以为其他人所用。

第三，这个函数对象没有数据成员、没有虚函数、没有显式声明的构造函数和析构函数，且对 operator() 的实现是内联的。用作 STL 比较器的函数对象一般都很小巧、简单且快速。当然可以设计一个具有重型实现的 STL 函数对象，但这种做法通常不是明智之举。当与 STL 协同使用时，在函数对象中避免（或尽量少）使用数据成员的另一个原因在于，STL 实现可能会为一个函数对象产生若干份复制，而且假定所有这些复制都是一致的。为了确保一个对象的所有复制一致，最简单的方式就是不要让对象带有任何数据成员。

现在我们就可以使用该函数对象对这个 union 进行排序：

```
sort( union, union+50, PopLess() );
```

请注意在这个 sort 调用中跟在 PopLess 后面的圆括号。PopLess 是一个类型，但是我们必须传入一个该类型的对象作为函数的参数。通过在 PopLess 类型名字后面附加一对圆括号，就创建了一个没名字的临时 PopLess 对象，此对象仅存活于函数调用期间（这个没名

字的对象即众所周知的“匿名临时对象”)。也可以声明并传入一个具名对象:

```
PopLess comp;
sort( union, union+50, comp );
```

然而,传入一个匿名临时对象更简单、更符合习惯,而且程序员击键次数更少。

使用函数对象作为比较器还有一个额外的好处,就是比较操作将被内联处理,而使用函数指针则不允许内联。原因在于,当一个sort函数模板实例化时,编译器知道比较器的类型是PopLess,从而使它知道PopLess::operator()将被调用,接着使它可以内联该函数,最后使它可以内联对嵌套的popLess函数的调用!

在STL中,函数对象另一个常见的用途是用作判断式^①。判断式是一个询问关于单个对象的真/假问题的操作(可以将比较器视作一种二元判断式)。

```
struct IsWarm : public std::unary_function<State,bool> {
    bool operator()( const State &a ) const
    { return a.aveTempF() > 60; }
};
```

73

STL判断式的设计指导方针与STL比较器的一致,惟一的例外在于,前者是一元函数,而非二元函数。从我们前面排过序的State结果开始,采用一个适当的判断式,可以让我们的生活变得温暖(warm)又悠闲:

```
State *warmandsparse = find_if( union, union+50, IsWarm() );
```

74

^①predicate, 判断式。另一个不当的说法(或译法)是“谓词”。——译者注

条款 21

重载与重写并不相同

重载和重写彼此之间没有任何关系。它们是两个完全不同的概念。对它们之间区别的不了解，以及对这两个术语马虎的使用，业已导致数不清的混乱和不计其数的 bug。

重载发生于同一个作用域内有两个或更多个函数具有相同的名字但签名不同时。函数的签名由它所声明的参数（或者说“形参”）的数目和类型构成。当编译器在一个作用域内查找一个函数名字时，发现不止一个函数具有该名字，它就会在该作用域的可用候选函数中，选择形参与函数调用中的实参有着最佳匹配的那一个函数（参见“成员函数查找[条款 24]”和“实参相依的查找[条款 25]”），此即为重载。

重写发生于派生类函数和基类虚函数具有相同的名字和签名时。在这种情况下，派生类函数的实现将会取代它所继承的基类函数的实现，以便满足对派生对象的虚拟调用。重写机制改变类的行为而不是改变其接口（另请参考“协变返回类型[条款 31]”）。

考虑如下简单的基类：

```
class B {  
    public:  
        //...  
        virtual int f( int );  
        void f( B * );  
        //...  
};
```

75

名字 `f` 在类 `B` 中被重载了，因为在同一个作用域内存在两个名字为 `f` 的不同的函数。（其中的重载加了灰色背景，表明它们是糟糕的代码。原因有二：也许并不希望重载一个虚函数，或者也可能并不希望在整型和指针类型之间进行重载。请分别参考 *C++ Gotchas* 和 *Effective C++*，以便了解缘由）

```
class D : public B {  
    public:  
        int f( int );  
        int f( B * );  
};
```

成员函数 `D::f(int)` 重写了基类中的虚函数 `B::f(int)`。成员函数 `D::f(B *)` 没有重写任何东西，因为 `B::f(B *)` 不是虚拟的。然而，它重载了 `D::f(int)`。注意，它并不重载基类成员 `B::f`，因为它们位于不同的作用域（参见“可选的关键字[条款 63]”）。

我再重复一遍，重载和重写是两个不同的概念，如果希望洞悉关于高级基类接口设计方面的建议和忠告，在技术上透彻理解这两个概念之间的区别是必不可少的。

条款 22

Template Method 模式

Template Method（模板方法）模式和 C++ 模板一点关系都没有。实际上，它是基类设计者为派生类设计者提供清晰指示的一种方式，这个指示就是“应该如何实现基类所规定的契约”（参见“多态[条款 2]”）。即使你认为这个模式应该换一个不同的名字，我还是请你继续使用标准名字“Template Method”。使用源自标准技术术语表的模式名字，会给你带来许多好处（参见“设计模式[条款 3]”）。

基类可以自由地通过其公有成员函数指定与外界的契约关系，并通过受保护的成员函数为派生类指明额外的细节。私有成员函数也可以用作类实现的一部分（参见“赋值和初始化并不相同[条款 12]”）。数据成员应该指定为私有，在下面的讨论中不再考虑它。

一个基类的成员函数是否应该为非虚拟的、虚拟的或纯虚拟的，这样的决策主要是基于该函数的行为如何被派生类定制。毕竟，使用基类接口的代码并不关心对象是怎么实现一个特定操作的，它只关心对一个对象执行该操作，而恰当地实现该操作则是对象自己的事情。

如果基类成员是非虚拟的，那么基类设计者就为以该基类为根所确立的层次结构指明了一个不变式（invariant）。派生类不应该用同名的派生类成员去隐藏基类非虚函数（参见“成员函数查找[条款 24]”）。如果不喜欢基类所指定的契约，可以（应该）去寻找另一个合乎心意的基类，而不要试图去改写基类的契约。

77

虚函数和纯虚函数指定的操作，其实现可以由派生类通过重写机制定制。一个非纯虚的函数提供了一个默认实现，并且不强求派生类一定要重写它，而一个纯虚函数则必须在具体派生类（即非抽象的派生类）中进行重写。这两种虚函数都允许派生类插入并取代其整个实现，同时保持接口不变。

Template Method 模式赋予基类设计者一种中级控制机制，该控制机制介于非虚函数提供的“占有它或离开它”和虚函数提供的“如果你不喜欢就替换掉所有东西”这两种机制之间。Template Method 确立了其实现的整体架构，同时将部分实现延迟到派生类中进行。通常来说，Template Method 被实现为一个公有非虚函数，它调用被保护的虚函数。派生类必须接受它所继承的非虚基类函数所指明的全部实现，同时还可以通过重写该公有函数所调

用的被保护的虚函数，以有限的方式来定制其行为。

```
class App {
public:
    virtual ~App();
    //...
    void startup() { // Template Method
        initialize();
        if( !validate() )
            altInit();
    }
protected:
    virtual bool validate() const = 0;
    virtual void altInit();
    //...
private:
    void initialize();
    //...
};
```

非虚拟的 startup Template Method 可以向下调用派生类提供的定制实现：

```
class MyApp : public App {
public:
    //...
private:
    bool validate() const;
    void altInit();
    //...
};
```

78

Template Method 是一个“好莱坞法则”的例子，即“不要 call 我们，我们会 call 你”（参见“Command 模式与好莱坞法则[条款19]”）。startup 函数的整体流程由基类决定，客户通过基类的接口来调用 startup，因此派生类设计者不知道 validate 或 altInit 何时会被调用。但他们知道当这两个方法被调用时，它们各自应该做什么。因此我们说，基类和派生类同心协力打造了一个完整的函数实现。

79

条款 23

名字空间

全局作用域日益变得拥挤不堪。每一个人连同他的亲朋好友都在使用相同的名字来实现不同库中的类和函数。例如，许多库都希望包括一个名为 `String` 的类。但是，如果使用两个不同的库，其中分别定义了一个 `String` 类型，就会得到多重定义错误甚至更糟糕的结果。存在各种各样与语言无关的途径可以解决这个问题（例如命名约定、预处理器……），但这些只会使事情变得更糟。在这里，名字空间（`namespace`）才是我们的“大救星”。

从某些方面来说，名字空间引入了复杂性（参见“实参相依的查找[条款 25]”），但名字空间的大多数用途都很简单。本质上，名字空间是对全局作用域的细分：

```
namespace org_semantics {  
    class String { ... };  
    String operator +( const String &, const String & );  
    // 其他类、函数、类型定义等……  
}
```

这个代码片断打开了一个名为 `org_semantics` 的名字空间，声明了一些有用的东西，然后采用闭花括号关闭该名字空间。总是可以向一个名字空间中加入更多的东西，重复上述过程即可。名字空间“对所有人”都是开放的。

可以注意到名字空间 `org_semantics` 中有些名字只有声明而没有定义。为了给这些名字提供定义，可以重新打开该名字空间：

```
namespace org_semantics {  
    String operator +( const String &a, String &b ) { // 哎呀!  
        //...  
    }  
}
```

81

作为一种替代方式，可以仅仅为该定义加上名字空间限定而无需重新打开名字空间：

```
org_semantics::String  
org_semantics::operator +(  
    const org_semantics::String &a,
```

```

    const org_semantics::String &b ) {
        //...
    }

```

这种方式有利于防止一不小心声明了一个新的名字空间(正如我们在第一个operator+定义中遗漏了对第二个参数进行const修饰那样)而不是定义一个已经声明的空间。无可否认,在这个例子中,看上去对org_semantics无尽地重复很让人厌烦,但这是为了安全而付出的代价!我们将讨论一些可以改善这种状况的途径。

如果希望使用定义于一个特定名字空间中的名字,必须告诉编译器该名字可以在哪一个名字空间中找到:

```

org_semantics::String s( "Hello, World!" );

```

尽管一些C++标准库组件还呆在全局作用域中(这些组件包括全局的operator new、operator delete、array new以及array delete^①等),然而大多数标准库组件现驻留在std(意指“standard(标准)”)名字空间中,因而对大多数标准库组件的使用,需要限定std名称:

```

#include <iostream>
#include <vector>
//...
void aFunc() {
    vector<int> a; // 错误! 我没看到哪儿定义了vector!
    std::vector<int> b; // 哦, 找到它了!
    cout << "Oops!" << endl; // 错误!
    std::cout << "Better!" << std::endl; // OK
    //...
}

```

显然,连续使用显式的限定符太乏味。缓解这种无趣状况的方式之一是使用using指令(using directive):

```

void aFunc() {
    using namespace std; // using 指令
    vector<int> a; // OK
    cout << "Hello!" << endl; // OK
}

```

^①array new和array delete意指用于数组分配和销毁的语法,也就是operator new[]和operator delete[]。参见条款37“数组分配”。这两个术语全书保留不译。——译者注

```
//...
}
```

从本质上来说，`using` 指令从名字空间中导入名字，使它们在该 `using` 指令的作用域内无需进行限定就可以被访问。在本例中，`using` 指令的作用域一直延伸到函数体末尾，后面再想使用该名字空间的东西，还需要进行明确的限定。正因为如此，许多 C++ 程序员（甚至很多还算有头脑的家伙）建议将 `using` 指令放在全局作用域中：

```
#include <iostream>
#include <vector>
using namespace std;
using namespace org_semantics;
```

这是个馊主意！现在我们基本上又退回到起点了。因为如此一来，名字空间中的所有名字在任何地方都可以被访问了，从而可能导致混淆和混乱。在头文件中这么做尤其糟糕，因为所有包含该头文件的文件都会受到这个糟糕决策的影响。在头文件中，我们通常坚持使用显式的限定，并且仅将 `using` 指令局限于较小的作用域中（例如函数体或函数体内的某个代码块），这样，它们的效用就会受到限制并易于控制。基本来说，在头文件中要坚持表现最好，在源文件中要表现得足够好，而在函数内大可放松一下。

使用 `using` 指令一个有趣的方面在于，它使得一个名字空间中的名字成为可用，但这种“可用”又不能算是绝对的可用，其实际效果相当于该名字空间中的名字被声明在全局作用域之中，而非局限于 `using` 指令所在的作用域中。所以说，如果 `using` 指令作用域中出现同名名字，就会隐藏该名字空间中的相应名字：

```
void aFunc() {
    using namespace std; // using 指令
    //...
    int vector = 12; // 一个欠佳的具名局部变量
    vector<int> a; // 错误！std::vector 被隐藏了
    std::vector<int> b; // OK，可以使用显式的限定
    //...
}
```

83

一个替代方式是使用 `using` 声明，它通过一个真正的声明提供对名字空间中名字的访问：

```
void aFunc() {
    using std::vector; // using 声明
    //...
    int vector = 12; // 错误！重新声明 vector
    vector<int> a; // OK
```

```

    //...
}

```

using 声明通常是介于冗长乏味的显式限定和不受限制地使用 **using** 指令之间的一种折衷。下面这种状况尤其适合：一个给定的代码段，只使用来自两、三个名字空间中有限的几个名字，但反复使用它们：

```

void aFunc() {
    using std::cout;
    using std::endl;
    using org_semantics::String;
    String a, b, c;
    //...
    cout << a << b << c << endl;
    // 等等.....
}

```

另一种对付冗长乏味的名字空间名字的方式是使用别名（**alias**）：

```

namespace S = org_semantics;

```

现在 **S** 可以用于替代 **org_semantics**（在别名的作用域内）。与 **using** 指令一样，最好避免在头文件中使用名字空间别名（毕竟 **S** 远比 **org_semantics** 更容易与其他名字冲突……）。

让我们瞥一眼匿名名字空间（**anonymous namespace**），来结束短暂的名字空间之旅：

```

namespace {
    int anInt = 12;
    int aFunc() { return anInt; }
}

```

这个名字空间的行为与以下名字空间的行为一致，当然，对于每一个匿名名字空间而言，都有一个惟一的“**__compiler_generated_name__**”：

```

namespace __compiler_generated_name__ {
    int anInt = 12;
    int aFunc() { return anInt; }
}
using namespace __compiler_generated_name__;

```

这是一个避免声明具有静态连接的函数和变量的新潮方式。在上面的匿名名字空间中，**anInt** 和 **aFunc** 都具有外部链接，但它们只能在其所出现的翻译单元（也就是预处理后的文件）内被访问，就像静态对象一样。

条款 24

成员函数查找

调用一个成员函数时，涉及三个步骤：第一步，编译器查找函数的名字；第二步，从可用候选者中选择最佳匹配函数；第三步，检查是否具有访问该匹配函数的权限。呃，就这么多。然而不可否认的是，每一步（尤其是前两步。参见“实参相依的查找[条款 25]”和“操作符函数查找[条款 26]”）都可能会很复杂，但从整体来看，函数匹配机制还是很简单的，不过区区三步而已。

很多与函数匹配有关的错误并非滋生于对编译器复杂的名字查找和重载函数匹配算法的误解，而是源于对这三大步简单而有序的性质的误解。考虑如下代码：

```
class B {
public:
    //...
    void f( double );
};
class D : public B {
    void f( int );
};
//...
D d;
d.f( 12.3 ); // 混淆
```

上面调用的是哪一个成员 `f`？让我们来分析一下。

步骤 1：查找函数的名字。因为我们正在调用一个 `D` 对象的成员，所以将从 `D` 的作用域开始查找并且立即定位到 `D::f` 上。

步骤 2：从可用候选者中选择最佳匹配函数。我们只有一个候选者 `D::f`，因此会尝试匹配该函数。可以通过将实参 `12.3` 从 `double` 转换为 `int` 而做到这一点（这是合法的，但通常不是我们想要的，因为那样会丢失精度）。

步骤 3：检查访问权限。我们（可能）会得到一个错误，因为 `D::f` 是私有成员。

基类中的哪一个看上去有着更好的匹配、并且可访问的函数 `f` 已经无关紧要，因为一

旦在内层作用域中找到一个，编译器就不会到外层作用域中继续查找该名字。内层作用域中的名字会隐藏外层作用域中相同的名字。在这一点，C++ 不同于 Java，在 Java 中，内层作用域中的方法名字和外层作用域中同名方法属于重载关系。

实际上，该名字甚至可以不是一个函数的名字：

```
class E : public D {  
    int f;  
};  
//...  
E e;  
e.f( 12 ); // 错误!
```

在这个例子中，我们得到一个编译期错误，因为在作用域 E 中查找名字 f，结果找到了一个数据成员，而不是成员函数。顺便说一下，这也是建立并遵从简单的命名习惯诸多理由之一。如果数据成员 E::f 被命名为 f_ 或 m_f，它就不会隐藏被继承的基类函数 f 了。

条款 25

实参相依的查找

名字空间对现代 C++ 编程和设计有着深远的影响（参见“名字空间[条款 23]”）。其中有些影响直接而明显，例如 `using` 声明和 `using` 指令以及采用名字空间作用域加以限定的名字。然而，名字空间还有一些在语法上不那么明显但仍然很基础、很重要的影响。实参相依的查找（Argument Dependent Lookup, ADL）就是其中之一。如同许多 C++ 特性一样，ADL 在有些场合下可能会变得很复杂，但在通常情况下，它的使用是相当直观的，并且解决的问题远比它所引入的要多。

ADL 背后蕴含的思想非常简单。当查找一个函数调用表达式中的函数名字时，编译器也会到“包含函数调用实参的类型”的名字空间中检查。举个例子，考虑如下的代码：

```
namespace org_semantics {
    class X { ... };
    void f( const X & );
    void g( X * );
    X operator +( const X &, const X & );
    class String { ... };
    std::ostream operator <<( std::ostream &, const String & );
}
//...
int g( org_semantics::X * );
void aFunc() {
    org_semantics::X a;
    f( a ); // 调用 org_semantics::f
    g( &a ); // 错误！调用具有歧义性
    a = a + a; // 调用 org_semantics::operator +
}
```

89

普通的查找是不会发现函数 `org_semantics::f` 的，因为它被嵌套在一个名字空间内，并且对 `f` 的使用需要以该名字空间的名字加以限定。然而，由于实参 `a` 的类型被定义于 `org_semantics` 名字空间中，因此，编译器也会到该名字空间中检查候选函数。

当然，像 ADL 这样复杂的规则也有叫你挠头的时候。对函数 `g`（接受一个指向 `org_semantics::X` 的指针）的调用就是一个佐证。在这个例子中，程序员本来以为编译器会

发现全局的 `g`，但由于实参的类型是 `org_semantics::X*`，因此该名字空间中的 `g` 也成了候选函数之一，从而导致这个调用具有歧义性。其实想一想，这种歧义未尝不是件好事，因为程序员本来也可能是希望调用 `org_semantics::g` 而非 `::g`。产生歧义使得事情变得更加明朗，程序员要么消除这个调用的歧义性，要么将其中一个函数的名字改掉。

注意，即使对 `g` 的调用导致了两个候选函数参与重载解析，`::g` 实际上也并未重载 `org_semantics::g`，因为它们不是声明于同一个作用域中的（参见“重载与重写并不相同[条款 21]”）。ADL 是关于函数如何被调用的一个属性，而重载则是关于函数被如何声明的一个属性。

可以看到 ADL 在对重载操作符的中缀调用中发挥的效用，例如在 `aFunc` 中对 `operator+` 的调用。在这里，中缀表达式 `a+a` 等价于 `operator+(a,a)`，ADL 将会在 `org_semantics` 名字空间中发现重载的 `operator+`（参见“操作符函数查找[条款 26]”）。

实际上，好多程序员广泛地使用了 ADL 却没意识到这一点。考虑如下对 `<iostream>` 的常见使用：

```
org_semantics::String name( "Qwan" );
std::cout << "Hello, " << name;
```

在这个例子中，对 `operator <<` 的第一个使用（即最左边的那一个），极可能调用的是类模板 `std::basic_ostream` 的一个成员函数，而第二个则是对位于 `org_semantics` 名字空间中重载的 `operator<<` 的非成员函数的调用。这些细节对于这段“欢迎”代码的编写者来说没什么关系，ADL 自觉地打理好了一切。

条款 26

操作符函数查找

有时看上去好像一个成员操作符函数重载了一个非成员的操作符，其实并非如此。这不是重载，而是不同的查找算法。考虑如下的类，它以成员函数的形式重载了一个操作符函数：

```
class X {
public:
    X operator %( const X & ) const; // 二元取余操作符
    X memFunc1( const X & );
    void memFunc2();
    //...
};
```

可以采用中缀或函数调用语法来调用这个重载的操作符函数：

```
X a, b, c;
a = b % c; // 采用中缀语法调用成员操作符 %
a = b.operator %( c ); // 成员函数调用语法
a = b.memFunc1( c ); // 另一个成员函数调用
```

当我们使用函数调用语法时，应用的是普通的查找规则（参见“成员函数查找[条款 24]”），也就是说，对 `b.operator%(c)` 调用的处理方式与 `memFunc1` 的相同。然而，对重载操作符的中缀调用的处理机制则与此不同：

```
X operator %( const X &, int ); // 非成员操作符
//...
void X::memFunc2() {
    *this % 12; // 调用非成员操作符 %
    operator %( *this, 12 ); // 错误！实参太多
}
```

对于中缀操作符调用来说，编译器不仅会考虑成员操作符，也会考虑非成员操作符（参见“实参相依的查找[条款 25]”），因此第一个对 `operator%` 的中缀调用，将会匹配非成员的那一个。这不是重载的实例，而是编译器在两个不同的地方查找候选函数。第二个对 `operator%` 的非中缀调用遵循标准的函数查找规则，因而匹配那个成员函数。这里我们遇到一

个错误，因为我们试图将三个实参传递给一个二元函数。（记住，对成员函数的调用存在一个隐式的实参 `this!`）

实际上，对重载操作符的中缀调用执行了一个退化形式的 ADL，即当确定将哪些函数纳入重载解析考虑范围时，中缀操作符中左参数的类（可能只有一个左参数，而没有右参数）的作用域和全局作用域都被考虑在内。ADL 则将这个过程扩展到被操作符实参所带入的其他名字空间中的候选操作符函数。注意这并不是重载。重载是一个和函数声明有关的静态属性（参见“重载与重写并不相同[条款 21]”），而 ADL 和中缀操作符函数查找都属于提供给函数调用的实参的属性。

条款 27

能力查询

在大多数情况下，当一个对象出现并开始工作时，它就能够执行我们需要它做的事情，因为它的能力在其接口中已被明确地通告。在这些情形下，不需要询问该对象是否能够胜任我们要它执行的工作，只管叫它去做好了：

```
class Shape {
    public:
        virtual ~Shape();
        virtual void draw() const = 0;
        //...
};
//...
Shape *s = getSomeShape(); // 获得一个 shape，并且叫它去……
s->draw(); // 干活！
```

即使不知道所面对的形状的精确类型，我们也知道该对象“是一个”Shape，因此它可以绘制（draw）自身。这很简单，也很高效，正是我们想要的效果。

然而，生活并非总是如此简单。有时一个出来准备工作的对象的能力并非那么显而易见。例如，我们可能需要一些形状是可以滚动的：

```
class Rollable {
    public:
        virtual ~Rollable();
        virtual void roll() = 0;
};
```

93

像 Rollable 这样的类通常称为“接口类（interface class）”，因为它只指定了接口，如同 Java 的接口一样。通常来说，这样的类没有非静态数据成员，没有声明构造函数。一个虚析构造函数和一个（或一组）纯虚函数指明了一个 Rollable 对象能够做什么。在这个例子中，我们等于是说，任何“是一个”Rollable 的对象都可以滚动（roll）。当然，一些形状可以滚动，另外一些则不行：

```
class Circle : public Shape, public Rollable { // 圆形可以滚动
    //...
```

```

    void draw() const;
    void roll();
    //...
};
class Square : public Shape { // 正方形则不能
    //...
    void draw() const;
    //...
};

```

当然，一些不是 Shape 类的对象也可能可以滚动：

```
class Wheel : public Rollable { ... };
```

在理想状况下，代码应该以这样的方式区分：在试图滚动（roll）它们之前，我们总是知道面对的对象是否为 Rollable，就像我们在试图绘制对象之前，就知道正在处理的是一个可以执行 draw 动作的 Shape 对象一样。

```

vector<Rollable *> rollingStock;
//...
for( vector<Rollable *>::iterator i( rollingStock.begin() );
    i != rollingStock.end(); ++i )
    (*i)->roll();

```

不幸的是，我们偶尔会碰到不知道一个对象是否具有所需能力的情形。在这种情形下，我们被迫执行一个能力查询。在 C++ 中，能力查询通常是通过对“不相关”的类型进行 dynamic_cast 转换而表达的（参见“新式转型操作符[条款 9]”）。

```

Shape *s = getSomeShape();
Rollable *roller = dynamic_cast<Rollable *>(s);

```

这种 dynamic_cast 用法通常称为“横向转型（cross-cast）”，因为它试图在一个类层次结构中执行横向转换，而不是向上或向下转换，如图 6 所示。

94

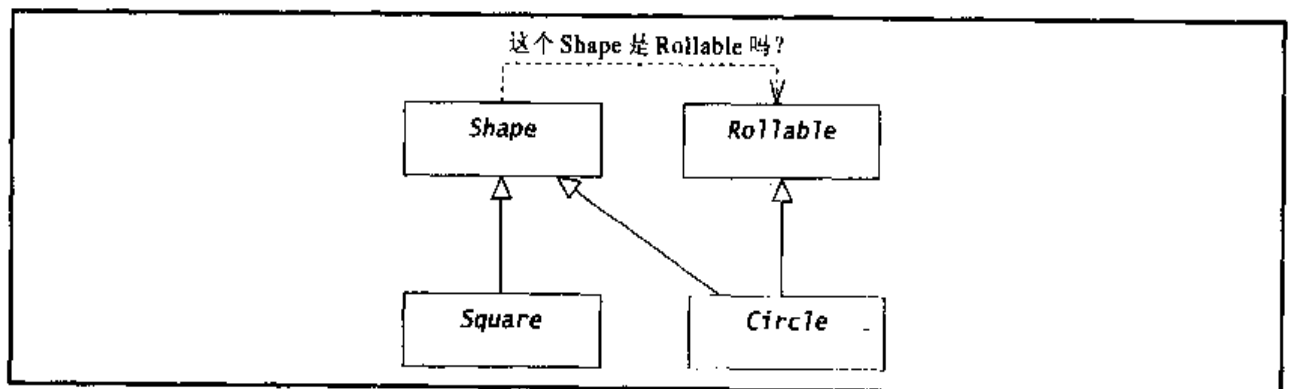


图 6 能力查询，“我能叫形状滚动吗？”

如果 `s` 实际指向的是一个 `Square`，那么 `dynamic_cast` 将会失败（结果产生一个空指针），从而知道 `s` 所指向的 `Shape` 不是 `Rollable`。如果 `s` 指向的是一个 `Circle` 或从 `Rollable` 派生下来的其他 `Shape`，那么转型将会成功，从而得知可以滚动这个形状。

```
Shape *s : getSomeShape();  
if( Rollable *roller = dynamic_cast<Rollable *>(s) )  
    roller->roll();
```

能力查询只是偶尔需要，但它们往往被过度使用。它们通常是糟糕设计的“指示器”。因此，除非找不到其他合理方式的困难境地，最好避免对一个对象的能力进行运行期查询。

条款 28

指针比较的含义

在 C++ 中，一个对象可以有多个有效的地址，因此，指针比较不是关于地址的问题，而是关于对象同一性的问题。

```
class Shape { ... };
class Subject { ... };
class ObservedBlob : public Shape, public Subject { ... };
```

在这个类层次结构中，ObservedBlob 同时派生于 Shape 和 Subject，并且由于是公有派生，因此存在从 ObservedBlob 到任一个基类的预定义转换。

```
ObservedBlob *ob = new ObservedBlob;
Shape *s = ob; // 预定义转换
Subject *subj = ob; // 预定义转换
```

存在这样的转换，意味着一个指向 ObservedBlob 的指针可以与指向其任何一个基类的指针进行比较。

```
if( ob == s ) ...
if( subj == ob ) ...
```

在这个例子中，这两个条件表达式的结果均为 true，即使 ob、s 和 subj 中包含的地址并不相同。考虑这些指针所指向的 ObservedBlob 对象的两种可能的内存布局，如图 7 所示。

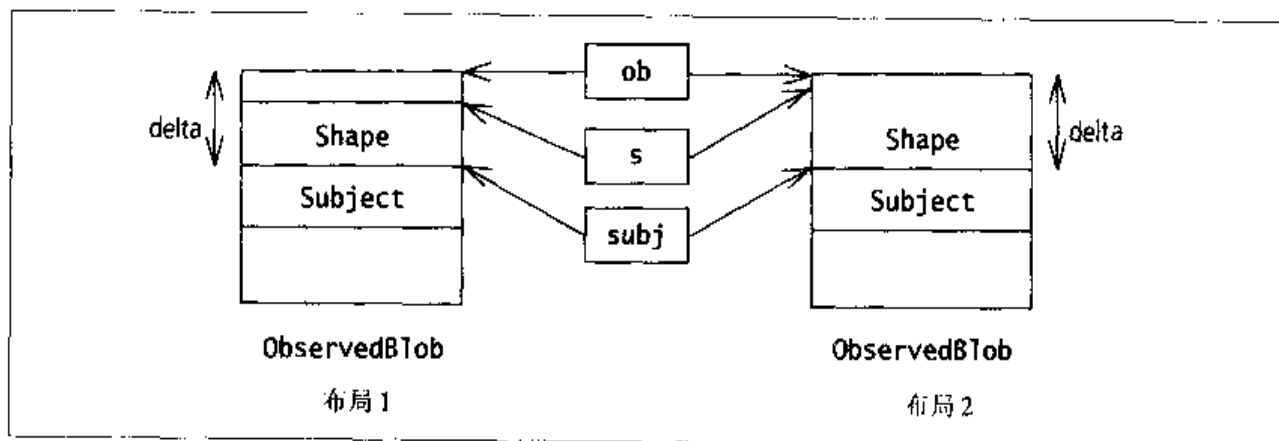


图 7 多重继承情形下两种可能的对象布局。不管是哪一种布局，对象都具有多个地址

在布局1中，s 和 subj 指向完整的对象内的 Shape 和 Subject 子对象，它们具有和 ob 所指向的完整对象所不同的地址。在布局2中，Shape 子对象恰巧与完整的 ObservedBlob 对象具有相同的地址，因此 ob 和 s 中包含相同的地址。

不管是哪种布局，ob、s 和 subj 都指向同一个 ObservedBlob 对象，因此编译器必须确保 ob 与 s 和 subj 的比较结果均为 true（不能拿 s 与 subj 进行比较，因为它们之间不具有继承关系）。编译器通过将参与比较的指针值之一调整一定的偏移量来完成这种比较。例如，表达式

```
ob == subj
```

可能被（不那么严格地）翻译为

```
ob ? (ob+delta == subj) : (subj == 0)
```

delta 是 Subject 子对象在 ObservedBlob 对象中的偏移量。换句话说，如果 ob 和 subj 都是空指针的话，它们就是相等的，否则 ob 被调整为指向其 Subject 基类子对象，然后再和 subj 进行比较。

从以上观察中，我们可以得到一个非常重要的经验：一般而言，当我们处理指向对象的指针或引用时，必须小心避免丢失类型信息。指向 void 的指针是常见的错误：

```
void *v = subj;  
if( ob == v ) // 不相等!
```

一旦通过将其复制到 void * 从而去掉 subj 中包含的地址的类型信息，编译器就别无他法，只好求助于原始地址比较（raw address comparison）了，而这样的比较对于指向类对象的指针来说，罕有是正确的。

条款 29

虚构造函数与 Prototype 模式

设想你现在身处一家瑞典餐馆，并且希望点餐。不幸的是，你的瑞典文水平仅仅局限于技术领域或“粗口”（一般是二者的结合）。菜单是用瑞典文写的，你看不懂，但是你注意到餐馆里对面的一位绅士正在享受美餐。于是，你把侍者叫过来，说道：

如果那位先生在吃鱼的话，我就点一份鱼。如果他在吃意大利面条的话，我就要一份意大利面条。如果他在吃鳗鱼的话，我就要一份鳗鱼。如果他在吃金橘蜜饯的话，我就点一份金橘蜜饯。

这听起来合理吗？当然不（别的先不说，你可能并不想在一家瑞典餐馆里点一份意大利面条）。这个“程序”存在两个基本的问题。首先，这种方式冗长乏味而且低效。其次，这种问询可能会以失败而告终。如果你把这一系列问题问完了，还是没能猜到那位先生在吃什么，结果会怎样？侍者将会离开，留下饥肠辘辘的你独自在那儿一筹莫展。退一步说，即使你碰巧知道菜单的全部内容，从而可以确保最终点餐成功，但如果你下一次去那家餐馆时，菜单已经被改掉了，你的问题列表就变得失效或不完善了。

显然，恰当的方式就是把侍者叫过来并且说，“我想要对面那位先生正在吃的东西。”

如果这位侍者拘泥于字面含义理解你的意思，他将会过去把那位先生吃了一半的食物夺下来并端到你的桌子上。然而，这样的行为很伤感情甚至可能会导致一场食物争夺战。当两个用餐者试图在同一时刻消费同样的食物时，就有可能发生这种不愉快的事情。如果侍者对自己的业务很熟悉，他将会端给你一份那位先生享用食物的“复制品”，且不会对被“复制”的食物状态造成任何影响。

99

有两个主要的原因需要使用“克隆”：你必须（或者更喜欢）对正在处理的对象的精确类型保持“不知情”，并且不希望改变被克隆的原始对象，也不希望受原始对象改变的影响。

在 C++ 中，提供了这种克隆对象的能力的成员函数，从传统上说，被称为“虚构造函数”。当然，并不存在什么虚构造函数，但是生成对象的一份复制品通常涉及到通过一个虚函数对其类的构造函数的间接调用，因此，即使不是真的虚构造函数，效果上也算是

虚构造函数了。最近，这种技术被称为 **Prototype** 模式的一个实例。

当然，我们必须了解所指向的那个对象。在这个例子中，只要知道所需的是一个 **Meal** 就可以了。

```
class Meal {
public:
    virtual ~Meal();
    virtual void eat() = 0;
    virtual Meal *clone() const = 0;
    //...
};
```

Meal 类型通过 **clone** 成员函数提供了克隆能力。**clone** 函数实际上是一种专门类型的 **Factory Method** 模式（参见“**Factory Method** 模式[条款30]”），它制造一个适当的产品，同时允许调用代码对上下文和产品类的精确类型保持不知情。派生于 **Meal** 的具体类（也就是那些实际存在的并且列在菜单上的 **Meal**）必须为纯虚 **clone** 操作提供一个实现：

```
class Spaghetti : public Meal {
public:
    Spaghetti( const Spaghetti & ); // 复制构造函数
    void eat();
    Spaghetti *clone() const
    { return new Spaghetti( *this ); } // 调用复制构造函数
    //...
};
```

100

（要想了解为何重写的派生类的 **clone** 函数的返回类型不同于基类对应函数的返回类型，请参考“**协变返回类型**[条款31]”。）

有了这个简单的框架，就可以生成任何类型的 **Meal** 的复制品，并且无需知道正在复制的 **Meal** 的实际类型的精确知识。注意，以下代码没有提及具体派生类，因而代码不会和任何当前（或将来出现）的 **Meal** 派生类相耦合。

```
const Meal *m = thatGuysMeal(); // 不管他正在吃什么东西……
Meal *myMeal = m->clone(); // 我都要一份和他一样的！
```

事实上，最终我们完全有可能点了一些从来没听过的食物。从效果上说，使用 **Prototype** 模式，对一个类型的存在不知情并不会对创建该类型的对象造成任何障碍。上面的多态代码可以通过编译并发布，并且如果以后增加新的 **Meal** 类型来增强功能，无需重新编译。

这个例子例证了软件设计中“不知情”的一些优点，尤其是在用于定制和扩展的框架结构软件设计中。记住：有些事情你知道的越少越好！

101

Factory Method 模式

一个高级设计通常要求基于一个现有对象类型来创建一个“适当”类型的对象。例如，我们可能拥有一个指向某种类型的 `Employee` 对象的指针或引用，现在需要为该类型的 `Employee` 生成一个适当类型的 `HRInfo` 对象^①，如图 8 所示。

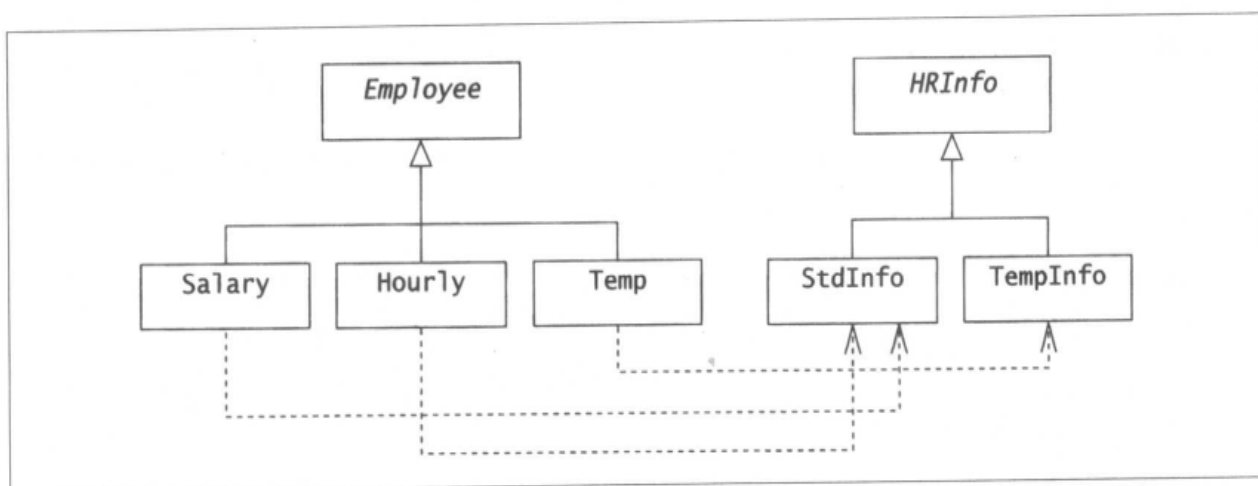


图 8 “伪平行”层次结构。应该如何将一个雇员与其对应的人力资源信息映射

这里，我们拥有两个几乎平行的 `Employee` 和 `HRInfo` 层次结构。`Salary` 和 `Hourly` 类型的雇员需要产生 `StdInfo` 对象，而 `Temp` 雇员则需要产生 `TempInfo` 对象。

高级设计方案非常简单：“为这个雇员创建一个适当类型的记录”。不幸的是，程序员通常将此类需求当成执行运行期类型查询的借口。换句话说，为了决定待产生的 `HRInfo` 对象的类型，实现此项需求的代码仅仅是询问一系列关于 `Employee` 的确切类型的问题。

一个常见、但总是错误的方式是使用“类型编码”和 `switch` 语句：

```
class Employee {
    public:
```

^① `HRInfo` 是 `Human Resources Information` 的缩写，意思是人力资源信息。——译者注

```

enum Type { SALARY, HOURLY, TEMP };
Type type() const { return type_; }
//...
private:
    Type type_;
    //...
};
//...
HRInfo *genInfo( const Employee &e ) {
    switch( e.type() ) {
        case SALARY:
        case HOURLY: return new StdInfo( e );
        case TEMP: return new TempInfo( static_cast<const Temp*>(e) );
        default: return 0; // 未知的类型编码!
    }
}

```

另外一种几乎同样糟糕的方式,是使用dynamic_cast来询问关于Employee对象一系列的私人问题:

```

HRInfo *genInfo( const Employee &e ) {
    if( const Salary *s = dynamic_cast<const Salary*>(&e) )
        return new StdInfo( s );
    else if( const Hourly *h = dynamic_cast<const Hourly*>(&e) )
        return new StdInfo( h );
    else if( const Temp *t = dynamic_cast<const Temp*>(&e) )
        return new TempInfo( t );
    else
        return 0; // 未知的employee类型!
}

```

这两种genInfo实现的主要缺点在于,它们与所有从Employee和HRInfo派生下来的具体类型相耦合,并且它们还必须熟悉从每一种Employee类型到其相应的HRInfo类型的映射。任一Employee、HRInfo或二者之间的映射发生了改动,都需要你去维护代码。在不同的开发组可能不断地向这两个继承层次结构中添加新类型或从中移除类型的情况下,这种代码维护工作几乎总是不能被正确地执行。另一个问题是,这两种方式都可能无法成功地识别Employee实参的确切类型,从而要求调用genInfo的代码提供对错误情况的处理。

根据上面的分析,正确的方式是去考虑从每一种Employee类型到对应的HRInfo类型的映射应该置于何处。换句话说,谁最清楚一个Temp employee需要何种HRInfo对象?理所当然,是Temp employee自己:

```

class Temp : public Employee {
public:
    //...
    TempInfo *genInfo() const
    { return new TempInfo( *this ); }
    //...
};

```

且慢，还有一个问题，我们可能不知道我们是在处理一个 Temp employee 而不是别的类型的 employee。这个问题很好解决，只要提供一个（纯）虚函数就可以了：

```

class Employee {
public:
    //...
    virtual HRInfo *genInfo() const = 0; // Factory Method
    //...
};

```

这是一个 Factory Method 模式的实例。实际上，我们不是向 employee 询问一系列生硬的私人问题，而是说“不管您是什么类型的 employee，请为自己生成适当的 HRInfo 对象”：

```

Employee *e = getAnEmployee();
//...
HRInfo *info = e->genInfo(); // 使用 Factory Method

```

105

Factory Method 的本质在于，基类提供一个虚函数“挂钩”，用于产生适当的“产品”。每一个派生类可以重写继承的虚函数，为自己产生适当的产品。实际上，我们具备了使用一个未知类型的对象（例如某种 Employee 对象）来产生另一个未知类型的对象（例如适当的 HRInfo 对象）的能力。

使用 Factory Method 模式通常意味着一个高级设计需要基于一个对象的确切类型产生另一个“适当”的对象，这样的需要往往发生于存在多个平行或几乎平行的类层次结构的情况下。Factory Method 模式通常是治疗一系列运行期类型查询问题的良方。

106

条款 31

协变返回类型

一般来说，一个重写的函数与被它重写的函数必须具有相同的返回类型：

```
class Shape {
public:
    //...
    virtual double area() const = 0;
    //...
};
class Circle : public Shape {
public:
    float area() const; // 错误！返回类型不同
    //...
};
```

然而，这个规则对于“协变返回类型 (covariant return type)”的情形来说有所放松。也就是说，如果 B 是一个类类型，并且一个基类虚函数返回 B*，那么一个重写的派生类函数可以返回 D*，其中的 D 公有派生于 B（即 D 是一个 (is-a) B）。如果基类虚函数返回 B &，那么一个重写的派生类函数可以返回一个 D &。考虑如下一个 shape 层次结构的 clone 操作（参见“虚构造函数和 Prototype 模式 [条款 29]”）：

```
class Shape {
public:
    //...
    virtual Shape *clone() const = 0; // Prototype (原型)
    //...
};
class Circle : public Shape {
public:
    Circle *clone() const;
    //...
};
```

重写的派生类函数被声明为返回一个 Circle * 而不是一个 Shape *。这是合法的，因为 Circle 是一个 Shape。注意，如果一个 Circle 被当作 Shape 进行操纵，从 Circle::clone

返回的 `Circle *` 就会被自动转换为 `Shape *` (参见“指针比较的含义[条款 28]”):

```
Shape *s1 = getACircleOrOtherShape();
Shape *s2 = s1->clone();
```

当直接操纵派生类型而不是通过其基类接口来操纵它们时,使用协变返回类型的优势就体现出来了:

```
Circle *c1 = getACircle();
Circle *c2 = c1->clone();
```

如果没有协变返回机制, `Circle::clone` 将不得不精确地匹配 `Shape::clone` 的返回类型,从而返回一个 `Shape *`。我们就被迫将返回结果转型为 `Circle *`。

```
Circle *c1 = getACircle();
Circle *c2 = static_cast<Circle *>(c1->clone());
```

再看另外一个例子。考虑如下 `Shape` 的 **Factory Method** 成员,它返回一个引用,指向与具体的形状对应的形状编辑器 (参见“**Factory Method** 模式[条款 30]”):

```
class ShapeEditor { ... };
class Shape {
public:
    //...
    virtual const ShapeEditor &
        getEditor() const = 0; // Factory Method
    //...
};
//...
class Circle;
class CircleEditor : public ShapeEditor { ... };
class Circle : public Shape {
public:
    const CircleEditor &getEditor() const;
    //...
};
```

在这个例子中,注意 `CircleEditor` 必须在 `Circle::getEditor` 的声明之前被完整地定义 (而不能仅仅加以声明)。因为编译器必须知道 `CircleEditor` 对象的布局,才能执行适当的地址操纵,从而将一个 `CircleEditor` 引用 (或指针) 转换为一个 `ShapeEditor` 引用 (或指针)。参见“指针比较的含义[条款 28]”。

协变返回类型的优势在于，总是可以在适当程度的抽象层面工作。如果我们是在处理 `Shape`，将获得一个抽象的 `ShapeEditor`；如果正在处理某种具体的形状类型，比如 `Circle`，我们就可以直接获得 `CircleEditor`。协变返回机制将我们从这样的一种处境解脱出来：不得不使用易于出错的转型操作来“重新”提供类型信息，而这种信息是一开始就不应该丢掉的：

```
Shape *s = getACircleOrOtherShape();
const ShapeEditor &sed = s->getEditor();
Circle *c = getACircle();
const CircleEditor &ced = c->getEditor();
```

条款 32

禁止复制

访问修饰符（包括 `public`、`protected` 和 `private`）可以用于表达和执行高级约束技术，指明一个类可以被怎样使用。

这些技术中最常见的一种是不接受对象的复制操作，这是通过将其复制操作声明为 `private` 同时不为之提供定义而做到的：

```
class NoCopy {
public:
    NoCopy( int );
    //...
private:
    NoCopy( const NoCopy & ); // 复制构造函数
    NoCopy &operator =( const NoCopy & ); // 复制赋值操作符
};
```

将复制构造函数和复制赋值操作符声明为 `private` 是必不可少的，否则编译器就会偷偷地将它们声明为公有、内联的成员。通过将其声明为 `private`，我们就谢绝编译器的干预，并确保对这两个操作的任何使用（不管是显式还是隐式）都会导致编译期错误：

```
void aFunc( NoCopy );
void anotherFunc( const NoCopy & );
NoCopy a( 12 );
NoCopy b( a ); // 错误！调用复制构造函数
NoCopy c = 12; // 错误！隐式调用复制构造函数
a = b; // 错误！调用复制赋值操作符
aFunc( a ); // 错误！传值，调用复制构造函数
aFunc( 12 ); // 错误！隐式调用复制构造函数
anotherFunc( a ); // 没问题！按引用传递
anotherFunc( 12 ); // 没问题
```

条款 33

制造抽象基类

抽象基类通常用于表示目标问题领域的抽象概念，创建这种类型的对象是没有什么意义的。我们通过至少声明一个纯虚函数（或者从别的类继承一个纯虚函数并且不予实现）使得一个基类成为抽象的，编译器将会确保无人能够创建该抽象基类的任何对象。例如：

```
class ABC {
public:
    virtual ~ABC();
    virtual void anOperation() = 0; // 纯虚函数
    //...
};
```

然而，有时找不到一个可以成为纯虚函数的合理候选者，但仍然希望类的行为像个抽象基类。在这些情形下，可以通过确保类中不存在公有构造函数来模拟抽象基类的性质。这意味着我们必须至少显式地声明一个构造函数，否则编译器将会隐式声明一个公有、内联的默认构造函数。而且，如果我们不显式地声明一个复制构造函数，编译器也会声明一个隐式的复制构造函数，因此，为了达到目的，通常必须显式地声明这两个构造函数：

```
class ABC {
public:
    virtual ~ABC();
protected:
    ABC();
    ABC( const ABC & );
    //...
};
class D : public ABC {
    //...
};
```

113

这两个构造函数被声明为受保护的，是为了既允许派生类的构造函数使用它们，同时阻止创建独立的 ABC 对象：

```
void func1( ABC );
void func2( ABC & );
```

```

ABC a; // 错误! 默认构造函数是受保护的
D d; // OK
func1( d ); // 错误! 复制构造函数是受保护的
func2( d ); // OK, 不涉及复制构造函数

```

另一种使一个类成为抽象基类的方式需要人为地将该类的一个虚函数指定为纯虚的。通常来说，析构函数是最佳候选者：

```

class ABC {
public:
    virtual ~ABC() = 0;
    //...
};
//...
ABC::~~ABC() { ... }

```

注意，在这个例子中，为该纯虚函数提供一个实现是必不可少的，因为派生类的析构函数将会隐式地调用其基类的析构函数（注意，从一个派生类析构函数内部对一个基类析构函数的隐式调用，总是非虚拟的）。

当一个类没有任何虚函数并且不需要显式声明构造函数时，适合用第三种方式。在这种情形下，采用受保护的、非虚的析构函数，乃是最佳实现方式：

```

class ABC {
protected:
    ~ABC();
public:
    //...
};

```

114

受保护的析构函数和受保护的构造函数发挥的效果基本相同，不过前者的报错发生于对象离开作用域时或被显式销毁时，而非对象创建时：

```

void someFunc() {
    ABC a; // 此时不会报错.....
    ABC *p = new ABC; // 此时不会报错.....
    //...
    delete p; // 错误! 析构函数是受保护的
    // 错误! 隐式调用 a 的析构函数
}

```

115

条款 34

禁止或强制使用堆分配

有时候，指明一些特定类的对象不应该被分配到堆（heap）上是个好主意。通常这是为了确保该对象的析构函数一定会得到调用。维护对本体对象（body object）的引用计数的句柄对象（handle object）就属于这种对象。具有自动存储区的类的局部对象，其析构函数会被自动地调用（通过 exit 或 abort 发生的非正常的程序终止情况除外），具有静态存储区的类的对象亦然（abort 情形除外），而堆分配的对象则必须被显式地销毁。

指明对象不应该被分配到堆上的方式之一，是将其堆内存分配定义为不合法：

```
class NoHeap {
public:
    //...
protected:
    void *operator new( size_t ) { return 0; }
    void operator delete( void * ) {}
};①
```

任何在堆上分配一个 NoHeap 对象的习惯性尝试，都将会导致编译期错误（参见“特定于类的内存管理[条款 36]”）：

```
NoHeap *nh = new NoHeap; // 错误！ NoHeap::operator new 是受保护的
//...
delete nh; // 错误！ NoHeap::operator delete 是受保护的
```

之所以给出 operator new 和 operator delete 的定义（和声明），是因为在一些平台上它们可能会被构造函数和析构函数隐式地调用。出于同样的原因，我们将其声明为 protected，因为它们可能会被派生类的构造函数和析构函数隐式地调用。如果 NoHeap 不打算用作基类，那么这两个函数也可以声明为 private。

同时，还要注意阻止在堆上分配 NoHeap 对象的数组（参见“数组分配[条款 37]”）。

^① 在上面的代码中，size_t 参数将被自动初始化为 NoHeap 对象的大小（以字节为单位）。而 void* 参数则被编译器自动设置为“将被 delete 的那个对象”的地址。——译者注

在这种情况下，只要将 `array new` 和 `array delete` 声明为 `private` 且不予定义即可，类似于禁止复制操作的方式（参见“禁止复制[条款 32]”）。

```
class NoHeap {
public:
    //...
protected:
    void *operator new( size_t ) { return 0; }
    void operator delete( void * ) {}
private:
    void *operator new[]( size_t );
    void operator delete[]( void * );
};
```

当然，在某些场合下，我们可能鼓励而非阻止使用堆分配，为此，只需将析构函数声明为 `private` 即可^①：

```
class OnHeap {
    ~OnHeap();
public:
    void destroy()
        { delete this; }
    //...
};
```

当对象的名字离开其作用域时，任何一个声明自动或静态 `OnHeap` 对象的尝试，都将会导致一个隐式的析构函数调用：

```
OnHeap oh1; // 错误！隐式调用私有析构函数
void aFunc() {
    OnHeap oh2;
    //...
    // 错误！隐式调用 oh2 析构函数
}
```

①同时要提供一个公有的销毁对象的方法（例如 `destroy`），否则创建的对象将无从销毁。——译者注

条款 35

placement new

直接调用构造函数是行不通的，然而，可以通过使用 placement new（定位 new）来“哄骗”编译器调用构造函数：

```
void *operator new( size_t, void *p ) throw()
{ return p; }
```

placement new 是 operator new 的一个标准的重载版本，也位于全局名字空间中，但是和我们通常所看到的 operator new 不同，语言明令禁止用户替换 placement new，而“普通的”operator new 和 operator delete 则可以被替换掉，只不过我们往往不应该那样做而已。placement new 的实现忽略了表示“大小”的实参，直接返回其第二个实参。placement new 允许我们在一个特定的位置“放置”对象，起到了调用一个构造函数的效果：

```
class SPort { ... }; // 表示一个串口
const int comLoc = 0x00400000; // 一个串口的空间位置
//...
void *comAddr = reinterpret_cast<void *>(comLoc);
SPort *com1 = new (comAddr) SPort; // 在 comLoc 位置创建对象①
```

区分 new 操作符和命名为 operator new 的函数很重要。new 操作符不可以被重载，所以其行为总是一样的。它调用一个名为 operator new 的函数，然后初始化返回的存储区。我们希望对内存分配行为进行任何改变，均需要通过 operator new 的不同重载版本实现，而非通过 new 操作符实现。同样的道理也适用于 delete 操作符和 operator delete。

①对

```
void *operator new( size_t, void *p ) throw()
```

而言，以下代码

```
SPort *com1 = new (comAddr) SPort;
```

size_t 参数被自动初始化为 SPort 对象的大小（以字节为单位），void * 类型的参数 p 则以 comAddr 为初值。——译者注

`placement new` 是函数 `operator new` 的一个版本，它并不实际分配任何存储区，仅仅返回一个（可能）指向已经分配好空间的指针。正因为调用 `placement new` 并没有分配空间，所以不要对其进行 `delete` 操作，记住这一点很重要。

```
delete com1; // 哎呀!
```

119

然而，尽管没有分配任何存储区，但确实创建了一个对象，这个对象应该在其生命周期结束时销毁。为此，应该避免使用 `delete` 操作符，代之以直接调用该对象的析构函数：

```
com1->~SPort(); // 调用析构函数而非 delete 操作符
```

然而，涉及直接析构函数调用的设计往往易于出错，常常会导致对同一个对象进行了多次析构或根本没有进行析构。因此，只有在必要时才使用这些设计，通常将其用于代码中被很好地隐藏和维护的地方。

也可以使用 `placement array new` 在给定的空间位置创建对象数组：

```
const int numComs = 4;
//...
SPort *comPorts = new (comAddr) SPort[numComs]; // 创建数组
```

当然，这些数组元素最终必须销毁：

```
int i = numComs;
while( i )
    comPorts[--i].~SPort();
```

对象数组可能出现问题的地方在于：当数组被分配时，每一个元素必须通过调用一个默认的构造函数而被初始化。考虑一个简单的、固定大小的缓冲区，可以向这个缓冲区 `append`（附加）新值：

```
string *sbuf = new string[BUFSIZ]; // 调用默认构造函数!
int size = 0;
void append( string buf[], int &size, const string &val )
    { buf[size++] = val; } // 刚才的默认构造动作白做了!
```

如果数组只有一部分元素被使用，或者元素被立即赋值，那么以上的做法就效率低下。更糟糕的是，如果数组的元素类型没有默认的构造函数，我们将会得到一个编译期错误。

`placement new` 通常用于解决此类缓冲区问题。采用这种方式，缓冲区占用的存储区的分配，可以避免被默认的构造函数初始化（如果默认构造函数中包含初始化代码）：

```
const size_t n = sizeof(string) * BUFSIZE;
```

```
string *ebuf = static_cast<string*> (::operator new( n ));  
int size = 0;
```

在第一次访问数组元素时，不能为其赋值，因为它还没有被初始化（参见“赋值和初始化并不相同[条款 12]”）。可以使用 **placement new** 通过复制构造函数来初始化元素：

```
void append( string buf[], int &size, const string &val )  
{ new (&buf[size++]) string( val ); } // placement new
```

通常，使用 **placement new** 也需要做一些清理工作：

```
void cleanupBuf( string buf[], int size ) {  
    while( size )  
        buf[--size].~string(); // 销毁已初始化的元素  
    ::operator delete( buf ); // 释放存储区  
}
```

这种方式快速而灵巧，它并不指望被普通大众所知晓。这一基本技术（以更高级的形式）广泛应用于大多数标准库容器的实现。

条款 36

特定于类的内存管理

如果不喜欢标准的 `operator new` 和 `operator delete` 对某个类类型的处置方式，不必郁闷地忍受。该类型完全可以拥有量身定制的 `operator new` 和 `operator delete`，以满足其特殊要求。

注意，我们无法对 `new` 操作符和 `delete` 操作符做什么，因为它们的行为是固定的，但可以改变它们所调用的 `operator new` 和 `operator delete`（参见“**placement new** [条款 35]”）。做这件事情的最佳方式是为类声明 `operator new` 和 `operator delete` 成员函数：

```
class Handle {
public:
    //...
    void *operator new( size_t );
    void operator delete( void * );
    //...
};
//...
Handle *h = new Handle; // 使用 Handle::operator new
//...
delete h; // 使用 Handle::operator delete
```

在一个 `new` 表达式中分配一个类型为 `Handle` 的对象时，编译器首先会在 `Handle` 的作用域内查找一个 `operator new`，如果没找到，它将会使用全局作用域中的 `operator new`。`operator delete` 的情形类似。因此，通常来说，如果定义了一个成员 `operator new`，最好同时也定义一个成员 `operator delete` 反之亦然。

成员 `operator new` 和 `operator delete` 是静态成员函数^①（参见“可选的关键字 [条款 63]”），这是有道理的。我们可以回想起静态成员函数没有 `this` 指针。由于这两个函

123

^①更确切地说，成员 `operator new` 和 `operator delete` 之所以是静态成员，是因为前者被调用于类对象构造之前，而后者则被调用于类对象析构之后，在这两种情形下，对象均不处于有效的状态，因此也就无 `this` 指针可用。——译者注

数仅仅负责获取和释放对象的存储区，因此它们用不着 `this` 指针。像其他静态成员函数一样，它们可以被派生类继承：

```
class MyHandle : public Handle {
    //...
};
//...
MyHandle *mh = new MyHandle; // 使用Handle::operator new
//...
delete mh; // 使用Handle::operator delete
```

当然，如果 `MyHandle` 已经声明了它自己的 `operator new` 和 `operator delete`，那么编译器将在查找期间首先发现并采用它们，而不再使用从基类 `Handle` 继承来的那个版本。

如果在基类中定义了成员 `operator new` 和 `operator delete`，要确保基类析构函数是虚拟的：

```
class Handle {
public:
    //...
    virtual ~Handle();
    void *operator new( size_t );
    void operator delete( void * );
    //...
};
class MyHandle : public Handle {
    //...
    void *operator new( size_t );
    void operator delete( void *, size_t ); // 注意第二个参数①
    //...
};
//...
Handle *h = new MyHandle; // 使用MyHandle::operator new
//...
delete h; // 使用MyHandle::operator delete
```

如果基类析构函数不是虚拟的，那么通过一个基类指针来删除一个派生类对象的结果就

①对于

`void operator delete(void *, size_t);`

而言，第二个类型为 `size_t` 的参数由编译器自动初始化为第一个参数所指对象的大小，以字节为单位。——译者注

是未定义的！一个实现可能简单地（并且很可能是不正确地）调用 `Handle::operator delete` 而不是 `MyHandle::operator delete`，但这样做发生任何事情都有可能。还要注意的，我们使用了一个带有两个参数的 `operator delete` 而不是普通的单参数版本的 `operator delete`。不过对于期望派生类来继承其 `operator delete` 实现的基类而言，这不过是另一个“普通”版本的成员 `operator delete` 而已。第二个参数用于保存正被删除的对象大小，这种信息在实现自定义内存管理时往往很有用。

一个常见的误解是以为使用 `new` 和 `delete` 操作符就意味着使用堆（或自由存储区）内存，其实并非如此。使用 `new` 操作符的惟一暗示是名为 `operator new` 的函数将被调用，且该函数返回一个指向某块内存的指针。没错，标准、全局的 `operator new` 和 `operator delete` 的确是从堆上分配内存，但成员 `operator new` 和 `operator delete` 可以做它们想做的任何事情。对于分配的内存到底从哪儿来没有任何限制：它可能来自一个特殊的堆，也可能来自一个静态分配的块，也可能来自一个标准容器内部，也可能来自某个函数范围的局部存储区。要说对于这个内存从哪儿来确实存在什么限制因素的话，那就是你的创造力和判断力了。例如，`Handle` 对象可以从一个静态块（static block）上进行分配，如下：

```
struct rep {
    enum { max = 1000 };
    static rep *free; // “自由”列表 (freelist) 的头部
    static int num_used; // 被使用的槽位 (slot) 数目
    union {
        char store[sizeof(Handle)];
        rep *next;
    };
};

static rep mem[ rep::max ]; // 静态存储区块 e
void *Handle::operator new( size_t ) {
    if( rep::free ) { // 如果 freelist 上有一些东西
        rep *tmp = rep::free; // 从 freelist 取出
        rep::free = rep::free->next;
        return tmp;
    }
    else if( rep::num_used < rep::max ) // 如果有剩余的槽位
        return &mem[ rep::num_used++ ]; // 返回未被使用的槽位
    else // 否则，我们现在所处的状况是……
        throw std::bad_alloc(); // 没有可用内存了！
}

void Handle::operator delete( void *p ) { // 添加到 freelist
```

```
static_cast<rep *>(p)->next = rep::free;  
rep::free = static_cast<rep *>(p);  
}
```

对此实现的一个“产品质量”的版本更要小心处理内存不足的情况，应该提供更强健的处理，并且要处理 `Handle` 的派生类型以及 `Handle` 数组的情况，等等。不过这段简单的代码已经表明 `new` 和 `delete` 不必非使用堆内存不可。

条款 37

数组分配

大多数C++程序员都知道在分配和归还内存时保持数组和非数组形式的操作符的匹配：

```
T *aT = new T; // 非数组
T *aryT = new T[12]; // 数组
delete [] aryT; // 数组
delete aT; // 非数组
aT = new T[1]; // 数组
delete aT; // 错误！应该采用数组形式的操作符
```

保持这些函数正确地成对使用很重要，因为数组的分配和归还所使用的函数不同于非数组形式。对于数组而言，new表达式不是使用operator new为数组分配存储区，而是使用array new。类似地，delete表达式也不是调用operator delete来释放数组的存储区，而是调用array delete^①。说得更确切一些，当分配一个数组时，使用一个不同的操作符，即new[]，而不是像分配一个非数组的对象那样使用new操作符，归还内存的情形类似。

array new和array delete分别是operator new和operator delete数组版本的对应物，它们的声明方式类似：

```
void *operator new( size_t ) throw( bad_alloc ); // operator new
void *operator new[]( size_t ) throw( bad_alloc ); // array new
void operator delete( void* ) throw(); // operator delete
void operator delete[]( void* ) throw(); // array delete
```

有关这些函数的数组形式最常出现的混乱情形，出现于一个特定的类或类层次结构使用成员operator new和operator delete，定义了自己的内存管理方式时（参见“特定于类的内存管理[条款36]”）：

```
class Handle {
public:
```

127

^①array new和array delete是针对数组分配和销毁的语法，也就是operator new[]和operator delete[]。这两个术语全书保留不译。——译者注

```

//...
void *operator new( size_t );
void operator delete( void * );
//...
};

```

Handle 类定义了非数组形式的内存管理函数，但它们并不被用于 Handle 数组的情形，对于数组的情形，调用的是全局 array new 和 array delete:

```

Handle *handleSet = new Handle[MAX]; // 调用::operator new[]
//...
delete [] handleSet; // 调用::operator delete[]

```

从逻辑上来说，只要声明了非数组形式的函数（即 operator new 和 operator delete），就应该为这些函数声明数组的形式，这是个好主意（然而奇怪的是，在日常编程实践中，这一点往往被人们所忽视）。如果目的是想调用全局的数组分配操作，那么，定义“仅仅转发对全局形式的调用”的局部形式，可以让事情变得更清晰：

```

class Handle {
public:
    //...
    void *operator new( size_t );
    void operator delete( void * );
    void *operator new[]( size_t n )
        { return ::operator new( n ); }
    void operator delete[]( void *p )
        { ::operator delete( p ); }
    //...
};

```

如果目的是不鼓励分配 Handle 数组，那么可以将数组形式的函数声明为私有的并且不提供定义（参见“禁止或强制使用堆分配[条款 34]”）。

另外一个混乱和错误之源在于，传递给 array new 的那个表示大小的参数值，取决于函数是如何被调用的。当 operator new 在一个 new 表达式中被（隐式地）调用时，编译器会决定需要多少内存，并将该数量作为参数传递给 operator new。这个数量就是正在分配的对象的大小：

```

aT = new T; // 调用 operator new( sizeof(T) );

```

也可以直接调用 operator new，在这种情况下，必须明确指明希望分配的字节数：

```

aT = static_cast<T *>(operator new( sizeof(T) ));

```

还可以直接调用 `array new`:

```
aryT = static_cast<T *>(operator new[]( 5*sizeof(T) ));
```

然而, 当通过 `new` 表达式隐式地调用 `array new` 时, 编译器常常会略微增加一些内存请求:

```
aryT = new T[5]; // 请求内存量为 5*sizeof(T) + delta 字节
```

所请求的额外空间一般用于运行期内存管理器 (`runtime memory manager`) 记录关于数组的一些信息, 这些信息 (包括分配的元素个数、每个元素的大小等) 对于以后回收内存是必不可少的。不过, 事情远没有这么简单, 编译器未必为每一个数组分配都请求额外的内存空间, 并且对于不同的数组分配而言, 额外请求的内存空间大小也会发生变化。

请求内存数量上的区别通常只在编写非常低层的代码时才需要考虑, 在这种情形下, 数组的存储区被直接处理。如果打算编写低层代码, 通常最简单的做法是避免直接调用 `array new` 以及编译器所执行的有关干预, 取而代之的是, 使用平凡朴素的 `operator new` (参见 “`placement new` [条款 35]”)。

条款 38

异常安全公理

编写异常安全的程序或库有点儿像在欧几里德几何学中证明定理。以尽可能小的一组公理为起点来证明一些简单的定理，然后再使用这些辅助定理去证明后续更复杂、更有意义的定理。处理异常安全问题与之类似：从异常安全的组件开始构建异常安全的代码（不过有一点值得一提，简单地将一组异常安全的组件或函数调用组合起来，并不能确保所得结果就是类型安全的。那样未免也太容易了一些，不是吗？）。然而，就像任何证明体系一样，最终必须建立一套异常安全的公理，我们依赖于这套公理来构建异常安全的结构。那么，都是些什么公理呢？

公理 1：异常是同步的

首先，异常是同步的并且只能发生于函数调用的边界。因此，诸如预定义类型的算术操作、预定义类型（尤其是指针）的赋值以及其他低层操作不会导致异常发生（它们可能会导致产生某种信号或中断，但这些东西都不是异常）。

操作符重载和模板使得情形变得复杂化了，因为通常很难判定一个给定的操作是否会导致一个函数调用并可能抛出异常。例如，如果对字符指针赋值，可以肯定不会抛出异常，但是如果对一个用户自定义的 String 进行赋值，就有可能导致发生异常：

```
const char *a, *b;
String c, d;
//...
a = b; // 不存在函数调用，不会抛出异常
c = d; // 函数调用，可能会抛出异常
```

[13]

对于模板而言，事情变得更不明朗：

```
template <typename T>
void aTemplateContext() {
    T e, f;
    T *g, *h;
    //...
```

```

    e = f; // 函数调用乎? 抛出异常乎?
    g = h; // 不是函数调用, 不会抛出异常
    // ...
}

```

由于存在这种不确定性, 因而模板内所有可能的函数调用都必须假定为就是函数调用, 这包括中缀操作符、隐式转换, 等等^①。

公理 2: 对象的销毁是异常安全的

该公理并非建立于技术基础之上, 而是建立于“社会”^②基础之上。按照惯例, 析构函数、`operator delete`以及`operator delete[]`不会抛出异常。设想有这么一个析构函数, 它必须 `delete` 两个指针数据成员。我们心里很清楚, 如果允许异常从析构函数传播出去的话, 将会受到批评、排斥, 为社会所不容, 因此可能忍不住去使用一个 `try` 语句块

```

X::~~X() {
    try {
        delete ptr1_;
        delete ptr2_;
    }
    catch( ... ) {}
}

```

这种做法是不必要的, 也是不可取的, 道理很简单, 这种对“为社会所不容”的畏惧心理同样也会影响到 `ptr1_` 和 `ptr2_` 所指向的对象的析构函数以及 `operator delete` 的作者。可以利用这种普遍的畏惧心理让代码变得更简洁一些:

```

X::~~X() {
    delete ptr1_;
    delete ptr2_;
}

```

132

①我看不出这段话与异常的同步性有什么紧密的关系, 也许作者奉行的“一本成功的书不是由书中的内容所构成的, 而是由该书省略的内容所构成的”的写作指导思想造成了这种状况。

归纳起来, 这段话的中心思想是函数调用可能会抛出异常。我对异常的同步性的理解是, 如果程序在某一点抛出了一个异常, 那么在该异常被处理之前程序将不会继续往下执行, 即程序执行流必须等待异常处理完成。是谓同步。

——译者注

②socially, 社会上的。书中多次出现这个词语, 意指 C++ 社群的看法、共识等。——译者注

公理 3：交换操作不会抛出异常

这同样是一个建立于 C++ 社群共识之上的公理，但它的公认度不如“禁止在析构函数中和销毁对象时抛出异常”那样来得广泛。乍看上去，交换（`swap`）不是一个太常见的操作，但在幕后它的使用很广泛，尤其是在 STL 的实现中。无论何时只要执行一个 `sort`、`reverse`、`partition` 以及其他许多操作，都会涉及到交换操作。一个异常安全的交换对于保证这些操作同样是异常安全的大有助益。参见“复制操作[条款 13]”。

[133]

条款 39

异常安全的函数

在编写异常安全的代码时，最困难的地方不在于抛出或捕获异常，而是在“抛出”和“捕获”之间我们应该怎么做。当一个被抛出的异常从 throw 表达式“奔向” catch 子句时，所经之路，任一个部分执行的函数在从执行堆栈上移除其激活记录之前，都必须清理它所控制的任何重要的资源。一般来说（但并非总是如此），编写一个异常安全的函数所需要的全部东西，不过是瞬间的自省和一些判断力而已。

举个例子，让我们考虑一下“赋值和初始化并不相同[条款 12]”中的 String 赋值实现：

```
String &String::operator =( const char *str ) {  
    if( !str ) str = "";  
    char *tmp = strcpy( new char[ strlen(str)+1 ], str );  
    delete [] s_;  
    s_ = tmp;  
    return *this;  
}
```

这个函数的实现看上去好像有点装饰过度了，可以取消使用临时变量从而采用更少行代码来实现它：

```
String &String::operator =( const char *str ) {  
    delete [] s_;  
    if( !str ) str = "";  
    s_ = strcpy( new char[ strlen(str)+1 ], str );  
    return *this;  
}
```

然而，尽管 array delete 有社会意义上的保证，即不抛出异常（参见“异常安全公理[条款 38]”），但是后面的 array new 并没有提供这样的承诺。如果在不清楚新缓冲区是否被成功分配之前就 delete 掉原来的缓冲区，可能就会使 String 对象处于糟糕的状态。Herb Sutter 在他的 *Exceptional C++* 一书中很好地总结了这种情形，在这里换个说法复述一

下：首先做任何可能会抛出异常的事情（但不会改变对象重要的状态），然后使用不会抛出异常的操作作为结束。在 `String::operator =` 的第一个实现中就是这么做的，如前所示。现在来看另一个例子，取自“**Command 模式与好莱坞法则**[条款 19]”：

```
void Button::setAction( const Action *newAction ) {
    Action *temp = newAction->clone(); // 先做了再说……
    delete action_; // 然后改变状态！
    action_ = temp;
}
```

由于它是一个虚函数，我们对调用 `clone` 所导致的异常相关的行为一无所知，因此我们做好最坏的假设，即，如果 `clone` 操作成功了，接下来进行的就是异常安全的 `delete` 和指针赋值操作，否则，从 `clone` 中抛出的一个异常将会导致从 `Button::setAction` 提前退出，且不会对任何人造成伤害。C++ 编程新手倾向于对这样的代码进行清理，从而导致其不再是异常安全的：

```
void Button::setAction( const Action *newAction ) {
    delete action_; // 先改变状态！
    action_ = newAction->clone(); // 然后……可能抛出异常？
}
```

在上例中，先于 `clone`（未提供异常安全的保证）执行 `delete` 操作（可以假定是异常安全的），若 `clone` 操作抛出一个异常，将会使得 `Button` 对象处于不一致的状态。

有必要提醒一下：编写正确的异常安全的代码其实很少使用 `try` 语句！一个新手在尝试编写异常安全的代码时，往往会使用不必要的甚至有害的 `try` 和 `catch` 语句块，从而把代码搞得乱七八糟：

```
void Button::setAction( const Action *newAction ) {
    delete action_;
    try {
        action_ = newAction->clone();
    }
    catch( ... ) {
        action_ = 0;
        throw;
    }
}
```

这个带有多余的 try 语句块和 catch 子句的版本，从这个意义上来说是异常安全的：如果 clone 抛出一个异常，那么 Button 对象将会保持稳固的状态^①（但很可能是不同的状态）。我们原先的那个版本更短小精悍，更简单，也更异常安全，因为倘若发生异常，Button 对象的状态不仅稳固，而且不会发生任何改变。

只要可能，尽量少用 try 语句块是一个好主意。主要在这种地方使用它们：确实希望检查一个传递的异常的类型，为的是对它做一些事情。在实践中，这些地方通常是代码和第三方的库之间、以及代码和操作系统之间的模块分界处。

137

①之所以说“Button 对象将会保持稳固的状态（但很可能是不同的状态）”，请对比代码

```
action_ = temp;
```

和

```
action_ = 0;
```

——译者注

条款 40

RAII

C++ 社群在赋予技术神秘的缩略语和古怪的名字方面有着悠久而自豪的传统。RAII 就是一个例子，它既古怪又神秘。RAII 表示“资源获取即初始化 (resource acquisition is initialization)”（而不是某些人会以为的“初始化即资源获取 (initialization is resource acquisition)”）。一句闲话，如果你打算搞怪，那就干脆怪到底，否则达不到效果）。

RAII 是一项很简单的技术。它利用 C++ 对象生命期的概念来控制程序的资源，例如内存、文件句柄、网络连接以及审计追踪 (audit trail) 等。RAII 的基本技术原理很简单。如果希望保持对某个重要资源的跟踪，那么创建一个对象，并将资源的生命期和对象的生命期相关联。如此一来，就可以利用 C++ 复杂老练的对象管理设施来管理资源。最简单的 RAII 形式是，创建这样的一个对象：其构造函数获取一份资源，而析构函数则释放这份资源：

```
class Resource { ... };
class ResourceHandle {
public:
    explicit ResourceHandle( Resource *aResource )
        : r_(aResource) {} // 获取资源
    ~ResourceHandle()
        { delete r_; } // 释放资源
    Resource *get()
        { return r_; } // 访问资源
private:
    ResourceHandle( const ResourceHandle & );
    ResourceHandle &operator =( const ResourceHandle & );
    Resource *r_;
};
```

ResourceHandle 对象的最美妙之处在于，如果它被声明为一个函数的局部变量，或作为函数的参数，或是一个静态对象，我们都可以保证析构函数会得到调用从而释放对象所引用的资源。当面对草率的维护或传播的异常时，如果希望保持对某项重要资源的跟

踪，此为一个可资利用的重要的属性。考虑以下未使用 RAII 的简单代码：

```
void f() {
    Resource *rh = new Resource;
    //...
    if( iFeelLikeIt() ) // 糟糕的维护
        return;
    //...
    g(); // 抛出异常?
    delete rh; // 我们一定能执行到这儿吗?
}
```

也许这个函数最初的版本是异常安全的，rh 所引用的资源总是可以得到释放。然而，随着时间的推移，当一些欠缺经验的维护人员往代码里插入一些可能会提前返回的代码、调用可能会抛出异常的函数、或者插入其他一些东西从而使得函数末尾的资源释放代码得不到执行的时候，这样的代码就不再是异常安全的了。使用 RAII，可以使函数变得更简单，并且更强健：

```
void f() {
    ResourceHandle rh( new Resource );
    //...
    if( iFeelLikeIt() ) // 没问题!
        return;
    //...
    g(); // 抛出异常乎? 没影响!
    // rh 析构函数执行 delete 操作
}
```

当使用 RAII 时，只有一种情况无法确保析构函数得到调用，就是当 ResourceHandle 对象被分配到堆上时，因为这么一来，只有显式地 delete 该 ResourceHandle 对象，此 ResourceHandle 对象所包含的对象的析构函数才会得到调用（让我揭露得更彻底一些。其实还有一些边缘性的情形，这包括调用 abort 或 exit 的情形，以及如果抛出的异常从未被捕获而导致不确定的情形）：

```
ResourceHandle *rhp =
    new ResourceHandle(new Resource); // 糟糕的主意!
```

140

RAII 技术在 C++ 编程中的影响是如此深远，以至于很难发现有哪个库组件或大型代码块未以某种风格使用它们。请注意，可以通过 RAII 进行控制的资源，范围非常广泛。除了控制本质上为内存块的资源（包括缓冲区、字符串、容器实现等诸如此类的东西）外，

还可以使用RAII来控制系统资源，包括文件句柄、信号量（semaphore）以及网络连接等，另外还包括一些不那么迷人的东西，例如登录会话（login session）、绘图形状甚至动物园里的动物。

考虑下面的类：

```
class Trace {
public:
    Trace( const char *msg ) : msg_(msg)
    { std::cout << "Entering " << msg_ << std::endl; }
    ~Trace()
    { std::cout << "Leaving " << msg_ << std::endl; }
private:
    std::string msg_;
};
```

在Trace这个例子中，控制的资源是一个准备打印的消息（当离开一个作用域时）。通过在不同类型的控制流程下监视其生命期来观察不同的Trace对象（包括自动的、静态的、局部的以及全局的等等）的行为，很有教育意义。

```
void f() {
    Trace tracer( "f" ); // 打印“Entering”消息
    ResourceHandle rh( new Resource ); // 获取资源
    //...
    if( iFeelLikeIt() ) // 没问题！
        return;
    //...
    g(); // 抛出异常乎？没影响！
    // rh析构函数执行delete操作！
    // tracer析构函数打印“Leaving”消息！
}
```

以上代码还展示了关于构造函数和析构函数结构激活的一个重要的定式：这些激活形成了一个栈。确切地说，先于rh声明并初始化tracer，这样就会保证rh将于tracer之前被销毁（即后初始化的先销毁）。推而广之，无论何时我们声明了一系列的对象，这些对象在运行期将会以特定的顺序被初始化，并且最终以相反的顺序被销毁。这个析构顺序不会发生变化，即使碰到了意外的return、传播的异常、不寻常的switch甚至邪恶的goto，均是如此（如果对这个宣称感到怀疑，我鼓励你去摆弄摆弄Trace类，我说过，它很有教育意义）。这个性质对于资源的获取和释放尤其重要，因为通常资源必须以特定的顺序进行获取且以相反的顺序进行释放。比方说，在发送一个审计消息之前，必须打开一个网络连接，而在该网络连接被关闭之前，必须发出一个关闭审计消息（closing audit message）。

这种基于栈的行为甚至延伸到了个体对象的初始化和析构方面。一个对象的构造函数按照其基类在继承列表中声明的顺序来初始化各个基类子对象 (base class subobject)，接着按照数据成员声明的顺序来初始化各数据成员，然后执行构造函数的本体。现在我们知道对象的析构函数的行为是怎样的了，就是“往回退着执行”。首先，执行析构函数本体，接着按与声明相反的顺序销毁对象的数据成员，最后是按与声明相反的顺序销毁对象的基类子对象。假如这样说还不明白的话，那么总得明白一点，对于对象获取所需资源并释放之而言，这种类似于栈的行为真的是非常方便。

new、构造函数和异常

为了编写完美的异常安全的代码，保持对任何分配的资源跟踪并且时刻准备着当异常发生时释放它们，是必不可少的。这个过程通常很简单。我们既可以将代码组织成无需回收资源的方式（参见“异常安全的函数[条款 39]”），也可以使用资源句柄（`resource handle`）来自动回收资源（参见“RAII[条款 40]”）。在极端的情况下，还可以采用“肮脏”的 `try` 语句块甚至嵌套的 `try` 语句块，但这应该视为一个例外，而不应被当成应该奉行的规则。

然而，关于 `new` 操作符的使用有一个很明显的问题。`new` 操作符实际上执行两个不同的操作（参见“placement new [条款 35]”）。首先，它调用一个名为 `operator new` 的函数来分配一些存储区，然后它调用一个构造函数来将未被初始化的存储区变成一个对象：

```
String *title = new String( "Kicks" );
```

其中的问题在于，如果发生了一个异常，我们说不清楚到底是 `operator new` 抛出来的，还是 `String` 构造函数抛出来的。区分这一点很重要，因为如果 `operator new` 成功了，而构造函数抛出异常，我们就应该调用 `operator delete` 来归还分配的（但未被初始化的）存储区。如果抛出异常的函数是 `operator new`，那么就没有发生任何内存被分配^①，因此我们就不应该调用 `operator delete`。

一种恐怖的方式是手工编写具有正确行为的代码，这是通过将分配和初始化行为分离并借助于一个 `try` 语句块完成的：

```
String *title // 分配原始存储区
    = static_cast<String *>(::operator new(sizeof(String)));
try {
    new( title ) String( "Kicks" ); // placement new
}
```

^①如果 `operator new()` 失败，将会抛出一个 `std::bad_alloc` 异常。——译者注

```
catch( ... ) {  
    ::operator delete( title ); // 如果构造函数抛出异常，则清理资源  
}
```

这段代码有那么多“错误”使得这种方式根本不值得考虑。以下的情形则会给你——过度操劳的程序员——带来更多的麻烦：如果 String 有一个 operator new 和一个 operator delete 成员的话（参见“特定于类的内存管理[条款36]”），这段代码将无法正确地工作。这是一个完美的反例，告诉我们，过于聪明的代码一开始或许可以工作，但在将来可能会因一些细微的改变（例如有人添加了特定于 String 内存管理机制的代码）而失败，而且失败的原因颇为微妙。

幸运的是，编译器可以帮我们处理这种情形，并且生成执行方式与上面手工编码方式相同的代码。不过有一个例外。它将会调用与执行分配任务的 operator new 相对应的 operator delete。

```
String *title = new String( "Kicks" ); // 使用成员 operator new——如果提供了  
的话  
String *title = ::new String( "Kicks" ); // 使用全局的 new/delete
```

特别地，如果分配时采用了成员 operator new，那么，如果 String 构造函数抛出了一个异常，则相应的成员 operator delete 将会被调用，进行存储区的回收工作。这是又一个好理由：如果声明了一个成员 operator new，那么最好也声明一个成员 operator delete。

从本质上来说，同样的情形也适用于数组分配以及使用了重载版本的 operator new[] 的其他分配，编译器将会试图发现并调用适当的 operator delete[]。

条款 42

智能指针

C++ 程序员都很忠诚。无论何时当面临需要一个语言所不支持的特性时，都不会抛弃 C++ 而移情别恋，而是会对 C++ 进行扩展以便获得感兴趣的特性。

举个例子，我们经常遇到这样的情形：需要某种东西的行为像个指针，但内建的指针又做不了我们想要做的事情。在这些情形下，C++ 程序员可以使用一个“智能指针”（参见“函数对象[条款 18]”，以便了解关于函数指针的类似的现象）。

智能指针是一个类类型，它乔装打扮成一个指针，但额外提供了内建指针所无法提供的能力。通常而言，一个智能指针使用类的构造函数、析构函数和复制操作符所提供的能力，来控制（或跟踪）对它所指向的东西的访问，而内建指针在这方面则无能为力。

所有智能指针都重载 `->` 和 `*` 操作符，从而可以采用标准指针语法来使用它们（一些罕见的智能指针甚至还重载了 `->*` 操作符，参见“指向类成员的指针并非指针[条款 15]”）。另有一些智能指针（尤其是用作 STL 迭代器的指针）还重载了其他一些指针操作符，包括 `++`、`--`、`+`、`-`、`+=`、`-=` 以及 `[]` 等（参见“指针算术[条款 44]”）。智能指针通常采用类模板来实现，从而使它们可以指向不同类型的对象。下面是一个非常简单的智能指针模板，它执行一个检查，确保在被使用之前不为空：

```
template <typename T>
class CheckedPtr {
public:
    explicit CheckedPtr( T *p ) : p_( p ) {}
    ~CheckedPtr() { delete p_; }
    T *operator ->() { return get(); }
    T &operator *() { return *get(); }
private:
    T *p_; // 我们所指向的东西
    T *get() { // 返回之前先检查指针是否为空
        if( !p_ )
            throw NullCheckedPointer();
    }
};
```

```

        return p_;
    }
    CheckedPtr( const CheckedPtr & );
    CheckedPtr &operator =( const CheckedPtr & );
};

```

智能指针的用法非常直观，酷似内建指针的用法：

```

CheckedPtr<Shape> s( new Circle );
s->draw(); // 效果与(s.operator ->())->draw()相同

```

在这个用法外表之下的关键点在于重载的`operator->`，此操作符必须被重载为一个成员函数，并且具有一个非同寻常的性质，即当它被调用时，它并不被消耗掉（consumed）^①。换句话说，当我们写`s->draw()`时，编译器识别出`s`不是一个指针，而是一个重载了`operator->`的类对象（即`s`是一个智能指针）。这将导致对成员重载的操作符的调用，本例中返回一个`Shape *`内建指针。然后该指针被用于调用`Shape`的`draw`函数。如果采用普通写法来编写代码，将会得到如下具有挑战性的表达式：`(s.operator->())->draw()`，其中包含了两个`->`操作符，一个是重载的版本，另一个是内建的版本。

除了重载`operator->`外，智能指针通常还重载`operator *`，从而使它们可用于指向不是类的类型，如下所示：

```

CheckedPtr<int> ip = new int;
*ip = 12; // 同ip.operator *() = 12
(*s).draw(); // 同样适用于指向类的指针

```

在C++编程中，智能指针有着普遍深入的使用，从资源句柄（参见“RAII [条款40]”和“`auto_ptr` 非同寻常 [条款43]”）到STL迭代器，从引用计数指针到指向成员函数的指针的包装器（wrapper），等等。一句话，无论在哪里，总有智能指针随时待命！

146

^①这里的“consumed”意思是，对`->`操作符而言，不会因为第一次调用就被“消耗掉”了，必要的话还可以继续使用。原因在于智能指针封装了一个指针成员，同时又重载了指针操作符`->`。 译者注

条款 43

auto_ptr 非同寻常

任何时候，当一个人讨论 RAII 时，总是有必要讨论一下 auto_ptr。这一向是一个必须履行的任务。提醒一下，倒不是我们对 auto_ptr 感到羞愧，但这的确有点儿像把你的兄弟介绍给别人：“他这个人其实挺好的，但是你要懂得欣赏他”。无可否认的是，你的兄弟和 auto_ptr 分别具有不同于常人和普通对象的世界观。

正如我们在“RAII [条款 40]”中讨论的那样，资源句柄是 C++ 编程中广为使用的技术，因此标准库提供了一个资源句柄模板，以便满足许多需用资源句柄的场合，这个模板就是 auto_ptr。auto_ptr 是一个类模板，用于生成具体的智能指针（参见“智能指针 [条款 42]”），它们知道在用完之后如何清理资源。

```
using std::auto_ptr; // 参见“名字空间[条款 23]”
auto_ptr<Shape> aShape( new Circle );
aShape->draw(); // draw 一个 circle
(*aShape).draw(); // 再 draw 一个 circle
```

就像所有有着良好设计的智能指针一样，auto_ptr 重载了 operator -> 和 operator *，因此通常可以假装自己在处理一个内建的指针。auto_ptr 有很多好处。首先，它非常高效。你不可能使用内建指针手工编写一个比 auto_ptr 性能还好的解决方案。其次，当 auto_ptr 离开作用域时，其析构函数将会释放它所指向（引用）的任何东西，正如手工编写的资源句柄所做的那样。在上面的代码片断中，aShape 所指的 Circle 对象将会有效地被垃圾收集。

auto_ptr 的第三个好处是，在类型转换方面，其行为酷似内建指针：

```
auto_ptr<Circle> aCircle( new Circle );
aShape = aCircle;
```

147

通过灵活使用模板成员函数（参见“成员模板[条款 50]”），一个 auto_ptr 可以复制给另一个，只要相应的内建指针支持这种操作即可。在上面的代码中，一个 auto_ptr<Circle> 可以赋值给一个 auto_ptr<Shape>，因为一个 Circle * 可以赋值给一个 Shape *（假定 Shape 是 Circle 的一个公有基类）。

其中 auto_ptr 不同于普通的智能指针（或普通对象）之处在于其复制操作。对于一般的类而言，复制操作（参见“复制操作[条款 13]”）不会影响到参与复制的源值。换句话说，如果 T 是某个类型

```
T a;
T b( a ); // 通过 a 复制构造 b
a = b; // 将 b 赋值给 a
```

那么，当采用 a 来初始化 b 时，a 的值不受影响，并且当 b 被赋值给 a 时，b 的值也不受影响。但对 auto_ptr 来说，情形并非如此！当我们将 aCircle 赋值给 aShape 时，参与赋值的源值和目标值都受到了影响。如果 aShape 是非空的，那么不管它指向（引用）的是什么东西，都将会被 delete 掉，并且代之以 aCircle 所指向的东西；除此之外，aCircle 也被设置为空。对于 auto_ptr 而言，赋值和初始化并不是真正的复制操作，它们实际上是将底层对象的控制权从一个 auto_ptr 转移到另一个 auto_ptr。可以将赋值或初始化操作的右参数视作“源”，而将左参数视作“接收器（sink）”。底层对象的控制权从“源”传递给“接收器”。对于资源句柄的情形来说，这是一个很好的属性。

然而，有两种常见的场合应该避免使用 auto_ptr。首先，它们永远都不应该被用作容器元素。容器中的元素通常在容器内部被拷来拷去，并且容器假定其元素遵从普通的非 auto_ptr 复制语义。这并不是说不可以使用智能指针作为容器元素，当然可以！只要它不是 auto_ptr 就行^①。其次，一个 auto_ptr 应该指向单个元素，而不应该指向一个数组。原因在于，当 auto_ptr 所指向的对象被删除时，它使用 operator delete 而非 array delete 来执行删除操作。如果 auto_ptr 指向的是一个数组，将会调用错误的“delete”操作符（参见“数组分配[条款 37]”）：

```
vector< auto_ptr<Shape> > shapes; // 糟糕的主意，很可能导致错误。
auto_ptr<int> ints( new int[32] ); // 糟糕的主意，目前尚无错误。
```

一般而言，对于指向数组的 auto_ptr 来说，标准库的 vector 或 string 是一个合理的替代品。

148

^①说“……只要它不是 auto_ptr 就行”是不确切的，关键是这个智能指针不可采用类似于 auto_ptr 所采用的“控制权移交”机制。——译者注

条款 44

指针算术

指针算术很直观。为了理解 C++ 中指针算术的性质，最好将指针放在数组的环境中考虑：

```
const int MAX = 10;  
short points[MAX];  
short *curPoint = points+4;
```

这段代码中有一个数组和一个指向该数组中间附近的指针，如图 9 所示。

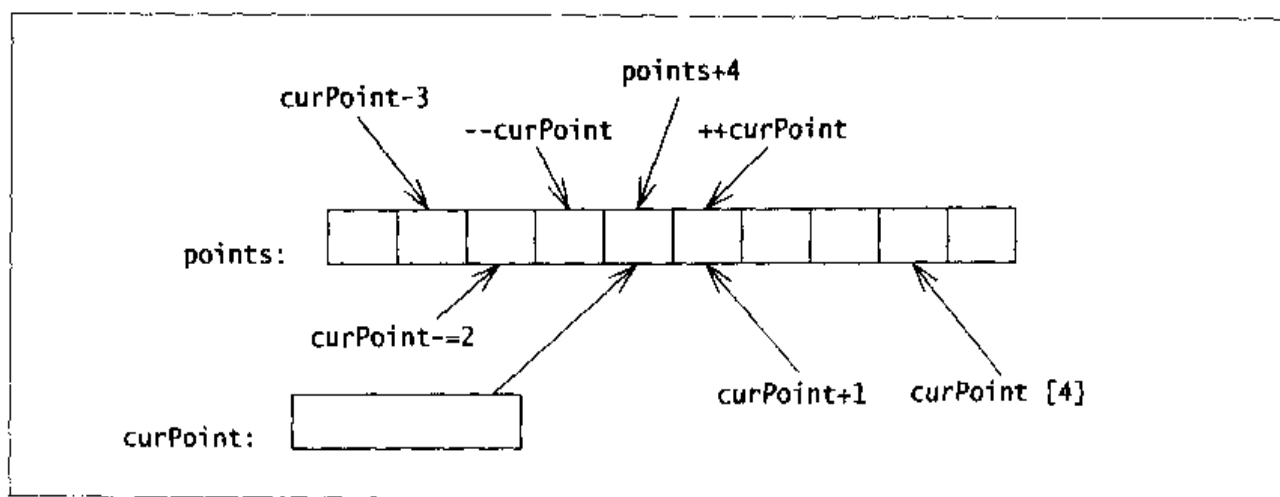


图9 对指针所包含的地址进行各种不同的算术操作

如果对 `curPoint` 执行递增或递减操作，等于是请求它指向 `points` 数组中的下一个或上一个 `short` 元素。换句话说，指针算术总是依照所指对象的大小比例进行的：将 `curPoint` 递增1并不是将指针地址增加一个字节，而是增加 `sizeof(short)` 个字节。这也正是 `void *` 不支持指针算术运算的原因，因为我们不知道某个 `void *` 所指向的对象的类型，所以无法正确地按比例进行指针算术运算。

这种简单的策略惟一可能招致混乱的情形发生于处理多维数组时，原因在于 C++ 编程新手往往会忘记多维数组其实是一个数组的数组：

```

const int ROWS = 2;
const int COLS = 3;
int table[ROWS][COLS]; // 一个具有 ROWS 个元素的数组，
                        // 其中每一个元素又是一个具有 COLS 个 int 元素的数组
int (*pTable)[COLS] = table; // 一个指针，指向具有 COLS 个 int 元素的数组

```

将一个二维数组形象地表示为图 10 所示的表会带来某些方便，但其实它在内存中的布局是线性的，如图 11 所示。

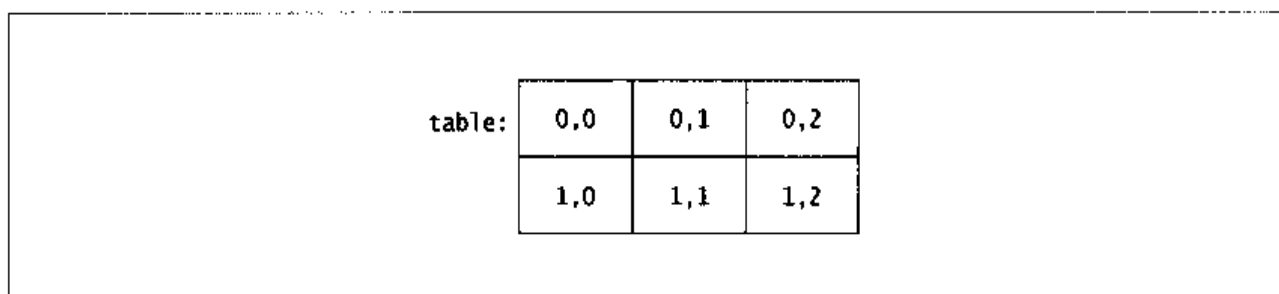


图 10 二维数组从概念上说是一个表

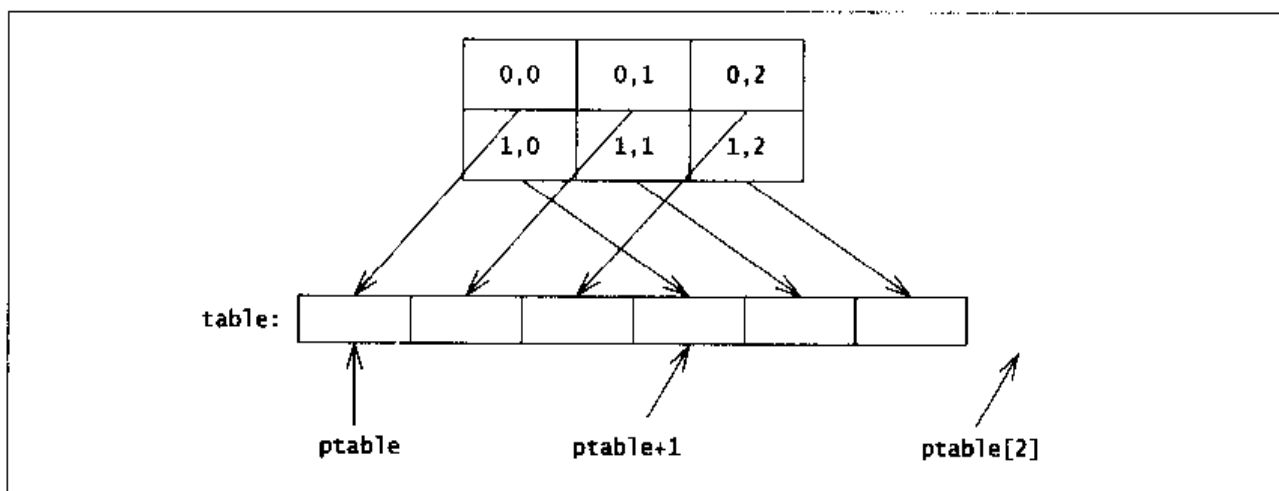


图 11 二维数组实际上是一个元素为一维数组的线性序列

对 `ptable` 执行指针算术时，一如既往，算术操作按照 `ptable` 所指对象的大小比例进行。不过，此时的对象是一个具有 `COLS` 个 `int` 元素的数组（其大小为 `sizeof(int)*COLS` 字节），而不是一个 `int`。

150

同一类型的两个指针可以进行减法运算，运算结果为参与运算的两个指针之间的元素个数（而不是它们之间的字节数）。如果第一个指针大于（即指向较高内存的地址）第二个指针，结果为正值，否则为负值。如果两个指针指向同一个对象，或者均为空，结果则为 0。两个指针相减的结果类型为标准 `typedef ptrdiff_t`，它通常是 `int` 的一个别名。两个指针之间不可以执行加法、乘法或除法，因为这些操作对于地址来说没有什么符合习

惯的意义。记住：指针并不是整数（另请参考“placement new [条款35]”）。

这种对指针算术通常意义上的理解，被 STL 迭代器用作设计隐喻（参见“STL[条款4]”和“智能指针[条款42]”）。STL 迭代器还允许指针风格的算术操作，即利用了和内建指针相同的语法的操作。实际上，内建指针与 STL 迭代器是相容的。考虑对 STL list 容器一个可能的实现，如图 12 所示。

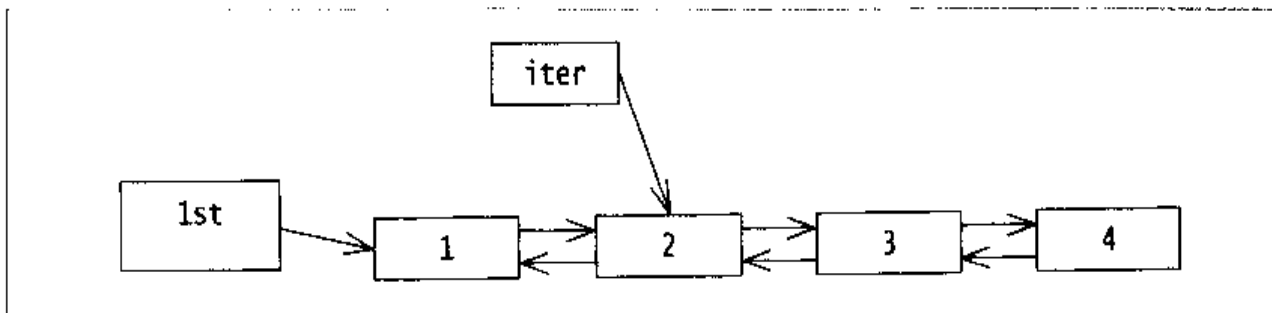


图 12 对标准 list 容器的一种可能的实现。list 迭代器并不是指针，但它模仿了一个指针

上图所示的配置结构可能是通过执行如下代码而产生的：

```
int a[] = { 1, 2, 3, 4 };
std::list<int> lst( a, a+3 );
std::list<int>::iterator iter = lst.begin();
++iter;
```

list 的迭代器不能是一个内建指针，而是一个带有重载操作符的智能指针。指针算术操作，例如 ++iter，并不是像递增一个指针那样来递增 iter，相反，它根据节点之间的链接关系，从 list 的当前节点移动到下一个节点。然而，这种算术操作酷似内建指针：递增操作使得迭代器移动到 list 中的下一个元素，就像对一个内建指针进行递增操作，使其移动到数组中的下一个元素。

条款 45

模板术语

精确地使用术语对于任何技术领域来说都很重要，对于程序设计领域尤其如此，对于 C++ 程序设计领域更是这样，而在 C++ 模板编程领域则达到极致。

图 13 描绘了 C++ 模板术语中最重要的方面。

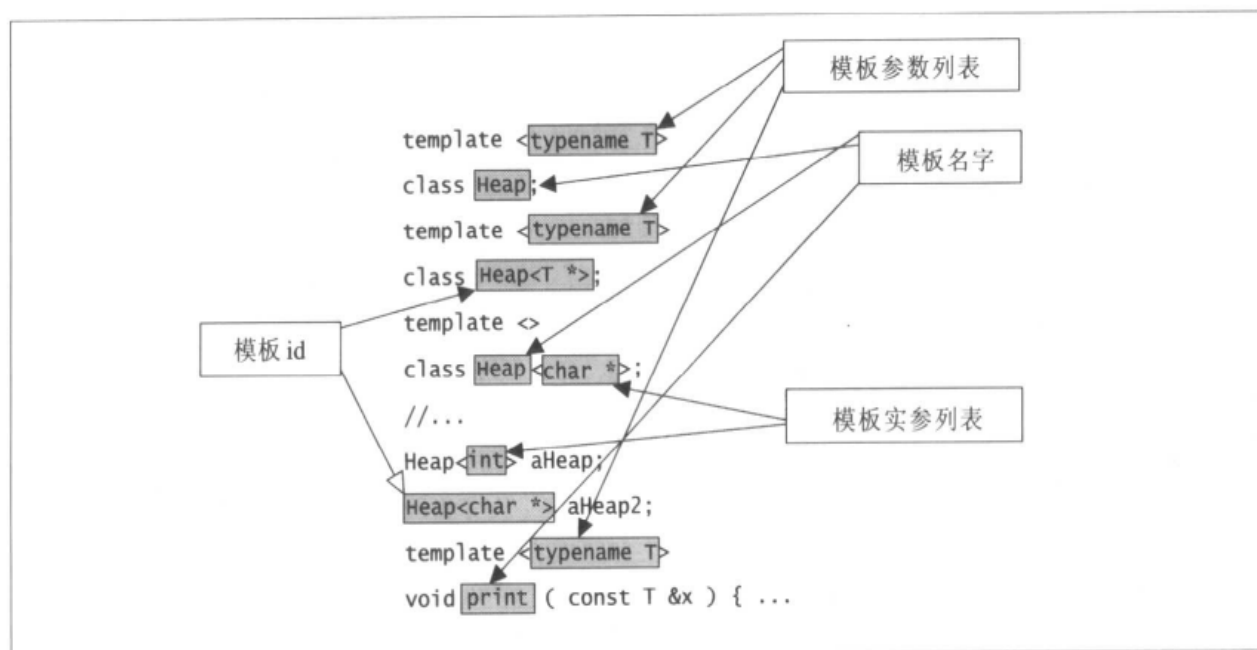


图 13 模板术语。精确地使用术语对于在模板设计中的精确沟通至关重要

尤其要小心区分模板参数 (template parameter) 和模板实参 (template argument)，前者用于模板的声明中，后者则用在模板的特化中：

```
template <typename T> // T 是一个模板参数
class Heap { ... };
//...
Heap<double> dHeap; // double 是一个模板实参
```

还要小心区分模板名字 (template-name) 和模板 id (template-id) 之间的区别，前者只是一个简单的标识符，后者则是指附带有模板实参列表的模板名称。

为数众多的 C++ 程序员将术语实例化 (instantiation) 和特化 (specialization) 混为一谈。对模板的特化是指当你以一套模板实参供应给一个模板时所得到的东西。特化可以显式进行, 也可以隐式进行。举个例子, 当我们写 `Heap<int>` 时, 我们是在使用 `int` 实参显式特化 `Heap` 类模板。当我们写 `print(12.3)` 时, 我们是在使用一个 `double` 实参隐式地特化 `print` 函数模板。模板的特化可能会也可能不会导致模板的实例化。例如, 如果存在一个针对 `int` 变量的自定义版本的 `Heap`, 那么 `Heap<int>` 特化会指向该版本, 从而不会发生实例化 (参见“类模板显式特化[条款 46]”)。然而, 如果使用了 `Heap` 主模板, 或者使用了它的一个局部特化版本 (参见“模板局部特化[条款 47]”), 那么将会发生实例化。

条款 46

类模板显式特化

类模板显式特化 (explicit specialization) 是很直观的。为了进行特化，首先需要—个通用的版本，这个通用的版本称为主模板：

```
template <typename T> class Heap;
```

主模板仅仅被声明为用于特化 (如上所示的 Heap)，但它通常也提供定义 (如下所示的 Heap)：

```
template <typename T>
class Heap {
public:
    void push( const T &val );
    T pop();
    bool empty() const { return h_.empty(); }
private:
    std::vector<T> h_;
};
```

这个主模板通过为有点挑战性的标准库堆算法 (heap algorithm) 提供一个易于使用的接口，从而实现一个堆数据结构。堆是一种线性化的树形结构，它对插入和检索操作进行了优化。将一个值压入一个堆中，实际上等于将该值插入到一个树形结构中；将一个值从堆中取出就等于移除并返回堆中的最大值。比方说，push 和 pop 操作可以分别使用标准算法 push_heap 和 pop_heap 进行实现：

```
template <typename T>
void Heap<T>::push( const T &val ) {
    h_.push_back(val);
    std::push_heap( h_.begin(), h_.end() );
}

template <typename T>

T Heap<T>::pop() {
    std::pop_heap( h_.begin(), h_.end() );
```

```

    T tmp( h_.back() );
    h_.pop_back();
    return tmp;
}

```

这个实现对于许多类型的值来说工作得挺好，但在处理指向字符的指针时碰了钉子。默认情况下，标准堆算法使用 < 操作符来对堆中的元素进行比较和组织。然而，对于指向字符的指针来说，这将会导致按照字符指针所指向的字符串的地址来组织堆，而不是按照字符串自身的值来组织堆。或者简单地说，堆将按照指针的值进行组织，而不是按照指针所指向的值进行组织。

我们可以提供一个针对指向字符的指针的显式特化版本（针对 Heap 主模板而言），来解决这个特别的难题：

```

template <>
class Heap<const char *> {
public:
    void push( const char *pval );
    const char *pop();
    bool empty() const { return h_.empty(); }
private:
    std::vector<const char *> h_;
};

```

其中的模板参数列表是空的，但要特化的模板实参则附加在模板名字后面。让人惊讶的是，这个类模板显式特化版本其实并不是一个模板，因为此时没有剩下任何未指定的模板参数了。出于这个原因，类模板显式特化通常被称为“完全特化”，以便与局部特化区分开，后者是一个模板（参见“模板局部特化[条款 47]”）。

这个领域的技术术语有点微妙。模板特化是一个提供了模板实参的模板名字（参见“模板术语[条款 45]”）。Heap<const char *>语法是一个模板特化，Heap<int>也是。然而，第一个对 Heap 的特化将不会导致对 Heap 模板的实例化（因为将会使用专为 const char * 定义的显式特化），但第二个特化将会导致对 Heap 主模板的实例化。

156

这个特化的实现可以根据 const char * 元素类型的需要进行定制。例如，push 操作可以基于指针所指向的字符串的值（而不是指针所包含的地址）将一个新值插入堆中：

```

bool strLess( const char *a, const char *b )
{ return strcmp( a, b ) < 0; }

void Heap<const char *>::push( const char *pval ) {
    h_.push_back(pval);
}

```

```
std::push_heap( h_.begin(), h_.end(), strLess );
}
```

注意，在 `Heap<const char *>::push` 的定义中没有出现 `template` 关键字，也没有出现参数列表，这不是一个函数模板，因为如上面所述，显式特化 `Heap<const char *>` 不是一个模板。

有了这个完全特化版本，我们就可以将针对 `const char *` 的 `Heap` 和其他 `Heap` 区分开了：

```
Heap<int> h1; // 使用主模板
Heap<const char *> h2; // 使用显式特化
Heap<char *> h3; // 使用主模板！
```

编译器根据主模板的声明来检查类模板特化。如果模板实参和主模板相匹配（对于 `Heap` 来说，就是如果只有单个类型名实参），编译器将会查找一个可以精确匹配模板实参的显式特化。如果希望除了提供一个针对 `const char *` 的 `Heap` 外，还希望提供一个针对 `char *` 的 `Heap`，就必须提供另外一个显式特化：

```
template <>
class Heap<char *> {
public:
    void push( char *pval );

    char *pop();
    size_t size() const;
    void capitalize();
    // 未提供 empty 函数！
private:
    std::vector<char *> h_;
};
```

157

注意，C++ 没有要求显式特化的接口必须和主模板的接口完全匹配。例如，在第一个针对 `const char *` 的 `Heap` 显式特化中，函数 `push` 的形参类型被声明为 `const char *` 而不是 `const char *&`。对于指针实参来说，这是一个合理的优化。在针对 `char *` 的 `Heap` 特化中，我们甚至走得更远，与主模板的接口差别更大。

我们添加了两个新函数，即 `size` 和 `capitalize`，这种做法是合法的，并且在某些情况下是非常有用的。但我们没有提供一个 `empty` 函数，这也是合法的，但通常是不可取的。当考虑类模板显式特化的接口时，将其与基类和派生类之间的关系进行类比是很有意义的（尽管类模板显式特化与类的派生之间不存在任何技术上的联系）。类层次结构的用户通常根据基类接口编写多态代码，并预期派生类将会实现该接口（参见“多态[条款 2]”）。类似地，用户通常根据主模板（如果已经定义并且声明了主模板）提供的接口

编写泛型代码，并预期任何特化版本将至少具有这些能力（像派生类那样，它也可以提供一些额外的能力）。考虑如下简单的函数模板：

```
template <typename T, typename Out>
void extractHeap( Heap<T> &h, Out dest ) {
    while( !h.empty() )
        *dest++ = h.pop();
}
```

158

如果下面的代码可以工作：

```
Heap<const char *> heap1;
//...
vector<const char *> vec1;
extractHeap( heap1, back_inserter(vec1) ); // 很好
```

而下面的代码无法通过编译

```
Heap<char *> heap2;
//...
vector<char *> vec2;
extractHeap( heap2, back_inserter(vec2) ); // 错误！未提供empty函数
```

159

那么该函数模板的作者将会抱怨“针对 char *”的Heap 显式特化的作者。

条款 47

模板局部特化

让我们开门见山，直奔主题：不能对函数模板进行局部特化。C++ 语言目前还不支持这个特性（尽管有朝一日或许支持）。所能做的就是重载它们（参见“重载函数模板[条款 58]”）。因此，在这个条款中，我们只考虑类模板。

类模板局部特化的工作方式很简单。就像完全特化一样，首先需要有一个通用的主模板来进行特化。让我们使用来自“类模板显式特化[条款 46]”中的 Heap 模板：

```
template <typename T> class Heap;
```

显式特化（俗称“完全特化”）用于以一套精确的实参来定制类模板。在“类模板显式特化[条款 46]”中，通过它来提供针对 `const char *` 和 `char *` 定制的 Heap 实现。然而，对于其他指针类型的 Heap 来说，仍然存在类似的问题。我们希望 Heap 依据指针元素所指向的值而非指针自身的值进行排序。

```
Heap<double *> readings; // 使用主模板，T 是 double *
```

由于 `double *` 类型与我们提供的两个针对字符指针的完全特化均不匹配，因此编译器将会实例化主模板。当然可以为 `double *` 提供完全特化，并可以根据需要为其他任何指针类型提供完全特化，但这种做法太麻烦，最终会导致代码难以维护。这其实是局部特化干的活：

```
template <typename T>
class Heap<T *> {
public:
    void push( const T *val );
    T *pop();

    bool empty() const { return h_.empty(); }
private:
    std::vector<T *> h_;
};
```

局部特化的语法类似于完全特化的语法，但它的模板参数列表是非空的。就像完全特

特化一样，类模板名字是一个模板 id 而不是一个简单的模板名字（参见“模板术语[条款 45]”）。

这个针对指针的局部特化允许适当地修改实现。例如，插入操作可以基于指针所指向的对象的值而非指针自身的值进行。首先，我们编写一个比较器，它根据指针所指向的值来比较两个指针（参见“STL 函数对象[条款 20]”）：

```
template <typename T>
struct PtrCmp : public std::binary_function<T *, T *, bool> {
    bool operator ()( const T *a, const T *b ) const
    { return *a < *b; }
};
```

现在我们使用这个“比较器”来实现具有正确行为的 push 操作：

```
template <typename T>
void Heap<T *>::push( T *pval ) {
    if( pval ) {
        h_.push_back(pval);
        std::push_heap( h_.begin(), h_.end(), PtrCmp<T>() );
    }
}
```

注意，和类模板的完全特化不同，局部特化是一个模板，在其成员的定义中，template 关键字和参数列表是不可缺少的。

和完全特化不同，这个版本的 Heap 的参数类型并没有被完全确定，它只是被部分地确定为 T *，而 T 是一个未指定的类型。这就是为什么说它是局部特化的原因。当使用任何（未经修饰的）指针类型来实例化一个 Heap 时，这个局部特化版将优先于主模板而被采用。进一步而言，当模板实参类型为 const char * 或 char * 时，针对 const char * 和 char * 的完全特化版本的 Heap 又将优先于这个局部特化而被采用。

162

```
Heap<std::string> h1; // 使用主模板，T 是 std::string
Heap<std::string *> h2; // 使用局部特化，T 是 std::string
Heap<int **> h3; // 使用局部特化，T 是 int *
Heap<char *> h4; // 使用针对 char * 的完全特化
Heap<char **> h5; // 使用局部特化，T 是 char *
Heap<const int *> h6; // 使用局部特化，T 是 const int
Heap<int (*)()> h7; // 使用局部特化，T 是 int ()
```

关于在各种不同的候选局部特化之间进行选择的完整规则，是相当复杂的，但大多数情况下这种选择还算比较直观。通常而言，最具体的、限制性最强的候选者将被选择。

局部特化机制非常精确，允许高精度地在候选者中进行选择。例如，可以向该局部特化集合中增加一个针对指向常量的指针的局部特化：

```
template <typename T>
class Heap<const T*> {
    //...
};
//...
Heap<const int*> h6; // 不同的局部特化，现在T是int
```

注意，正如我们在“类模板显式特化[条款 46]”中讨论的那样，编译器根据主模板的声明来检查类模板特化。如果模板实参和主模板相匹配（对于 Heap 来说，就是如果只有单个类型名字实参），编译器将查找和模板实参有着最佳匹配的完全特化或局部特化。

其中有一点很微妙但有用：主模板的完全特化或局部特化必须采用与主模板相同数量和类型的实参进行实例化，但它的模板参数列表并不需要具有和主模板相同的形式。对于 Heap 来说，主模板带有单个类型名字参数，因此 Heap 的任何完全特化或局部特化都必须采用单个类型名字实参来实例化：

```
template <typename T> class Heap;
```

163

所以，Heap 的完全特化仍然带有单个类型名字模板实参，但模板参数列表不同于主模板的模板参数列表，因为完全特化的模板参数列表是空的。

```
template <> class Heap<char*>;
```

Heap 的局部特化也必须带有单个类型名字模板实参，并且在其模板头部，模板参数列表也可以带有单个类型名字参数：

```
template <typename T> class Heap<T*>;
```

但它未必非得如此：

```
template <typename T, int n> class Heap<T[n]>;
```

当使用一个数组类型来特化 Heap 时，将会选用这个局部特化，例如：

```
Heap<float*[6]> h8; // 使用局部特化，T是float*且n为6
```

本质上，这个局部特化等于在说，“这个局部特化和主模板一样，带有单个类型参数，但该参数必须是一个具有 n 个类型为 T 的元素的数组”。考虑更棘手的一些例子：

```
template <typename R, typename A1, typename A2>
class Heap<R (*)(A1,A2)>;
```

```
template <class C, typename T>
class Heap<T C::*>;
```

有了这些额外的局部特化，就可以采用“指向带有两个参数的非成员函数”的指针对 Heap 进行特化，以及采用指向数据成员的指针进行特化（尽管为何需要对这些东西使用 Heap 只是一个猜测）：

```
Heap<char *(*)(int,int)> h9;           // 使用局部特化
                                         // R是char *, A1和A2是int
Heap<std::string Name::*> h10;         // 使用局部特化
                                         // T是string, C是Name
```

类模板成员特化

关于类模板显式特化和局部特化一个常见的误解是，一个特化版本会以某种方式从主模板那里“继承”一些东西。事实并非如此。对于主模板而言，类模板的完全特化或局部特化全然是单独的实体，它们不从主模板“继承”任何接口或实现。然而，从非技术的意义来说，一个特化版本确实从主模板那里继承了一套有关接口和行为的“期望”，因为用户在根据主模板的接口编写泛型代码时，通常期望所编写的代码同样可以处理特化的情形。

这就意味着一个完全特化或局部特化通常必须重新实现主模板具备的所有能力，即使只有部分实现需要定制也得如此。一个替代的方式是只去特化主模板成员函数的一个子集。举个例子，考虑 Heap 主模板（参见“类模板显式特化[条款 46]”）：

```
template <typename T>
class Heap {
public:
    void push( const T &val );
    T pop();
    bool empty() const { return h_.empty(); }
private:
    std::vector<T> h_;
};
```

针对 `const char *` 的完全特化 Heap 替代了主模板的全部实现，其实对于字符指针的堆来说，主模板 Heap 的私有成员和 `empty` 成员函数已经完全够用了，我们真正要做的全

部事情就是特化 `push` 和 `pop` 成员函数：

```
template <>
void Heap<const char *>::push( const char *const &pval ) {
    h_.push_back(pval);
    std::push_heap( h_.begin(), h_.end(), strLess );
}
```

```

template<>
const char *Heap<const char *>::pop() {
    std::pop_heap( h_.begin(), h_.end(), strLess );
    const char *tmp = h_.back(); h_.pop_back();
    return tmp;
}

```

这些函数是对 Heap 主模板相应成员的显式特化，并且将用于替换针对 Heap<const char*>的隐式实例化的版本。

注意，这些函数的接口必须和“它们正在特化其成员”的模板的相应接口精确匹配。例如，主模板将 push 声明为带有一个类型为 const T & 的参数，因此针对 const char * 的 push 显式特化的实参类型必须为 const char * const &（即一个引用，指向一个 const 指针，后者又指向一个 const char）。可以注意到，在从整体上提供 Heap 模板的完全特化时，并没有这个限制，其中 push 的实参只是简单地声明为 const char *。

为了考察更棘手的情况（当采用模板编程时这是很常见的事情），让我们考虑，如果针对一般指针的局部特化 Heap 已经可用了，将会发生些什么（参见“模板局部特化[条款 47]”）：

```

template <typename T>
class Heap<T *> {
    //...
    void push( T *pval );
    //...
};

```

166

如果这个 Heap 局部特化已经存在，那么对 push 的显式特化就必须符合该局部特化中 push 成员的接口，因为该函数相当于针对 Heap<const char *>进行实例化所得的结果。所以，显式特化现在必须声明为：

```

template <>
void Heap<const char *>::push( const char *pval ) {
    h_.push_back(pval);
    std::push_heap( h_.begin(), h_.end(), strLess );
}

```

最后指出两点：首先，除了成员函数外，类模板的其他成员可以被显式特化，这包括静态成员和成员模板。

其次，显式特化和显式实例化之间的区别常常混淆。正如我们在本条款中看到的那

样，显式特化是为模板或模板成员提供定制版本的一种手段，这种定制版本不同于对主模板的隐式实例化所得到的东西。显式实例化仅仅是明确地告诉编译器去实例化一个成员，所得结果与隐式实例化所得的东西是一致的：

```
template void Heap<double>::push( const double & );
```

参见“只实例化要用的东西[条款 61]”。

条款 49

利用 typename 消除歧义

即便经验丰富的 C++ 程序员也常常会被相当复杂的模板语法搞得垂头丧气。在所有的语法折磨中，没有比偶尔需要帮助编译器消除一个解析歧义更让人晕头转向的了。

作为一个例子，让我们来看一个简单的非标准容器模板实现的一部分：

```
template <typename T>
class PtrList {
public:
    //...
    typedef T *ElemT;
    void insert( ElemT );
    //...
};
```

对于类模板来说，以嵌套类型名字嵌入关于它们自身的信息，是一个常见的编程实践。这允许我们通过适当的嵌套名字来访问有关实例化的模板的信息（参见“嵌入的类型信息[条款 53]”和“traits[条款 54]”）：

```
typedef PtrList<State> StateList;
//...
StateList::ElemT currentState = 0;
```

嵌套的名字 ElemT 允许我们很容易地访问被 PtrList 模板认可的元素类型。如果采用类型名字 State 来实例化 PtrList，那么元素类型就是 State *。在其他情况下，PtrList 可能采用智能指针元素类型来实现，一个精密复杂的 PtrList 实现可能会基于用来实例化它的类型的属性，来改变其实现（参见“针对类型信息的特化[条款 52]”）。使用嵌套的类型名字，有助于将 PtrList 用户与 PtrList 内部实现决策之间隔离开来。

169

下面是另外一个非标准容器：

```
template <typename Etype>
class SCollection {
public:
    //...
```

```

typedef Etype ElemT;
void insert( const Etype & );
//...
};

```

看上去 SCollection 的设计使用了与 PtrList 相同的一套命名标准，因为它也定义了一个嵌套的 ElemT 类型名字。坚持使用一套已经确立的约定是很有意义的，因为（除了其他优点外）这样做允许我们编写可以与某个范围内的不同容器类型协作的泛型算法。例如，我们可以编写一个简单的工具算法，它采用具有适当元素类型的数组的内容来填充与之相容的容器：

```

template <class Cont>
void fill( Cont &c, Cont::ElemT a[], int len ) { // 错误!
    for( int i = 0; i < len; ++i )
        c.insert( a[i] );
}

```

不幸的是，我们会得到一个语法错误。嵌套的名字 Cont::ElemT 并不被识别为一个类型名字！其中的问题在于，在 fill 模板的上下文中，编译器没有足够的信息来决定嵌套的名字 ElemT 到底是一个类型名字，还是一个非类型的名字。C++ 标准约定，在这种情形下，嵌套的名字被假定为一个非类型的名字。

乍看上去这对你没什么意思，这很正常，有这种感觉的不只你一个。然而，让我们在一个不同的上下文中看看编译器能获得一些什么样的信息。首先考虑有一个非模板的类的情形：

```

class MyContainer {
public:
    typedef State ElemT;
    //...
};
//...
MyContainer::ElemT *anElemPtr = 0;

```

170

其中显然没有任何问题，因为编译器可以检查 MyContainer 类的内容，核实它确实有一个名为 ElemT 的成员，并且注意到 MyContainer::ElemT 实际上是一个类型名字。对于从类模板生成的类来说，事情同样简单：

```

typedef PtrList<State> StateList;
//...
StateList::ElemT aState = 0;
PtrList<State>::ElemT anotherState = 0;

```

对于编译器来说，一个实例化的类模板其实就是一个类，因此对类类型 `PtrList<State>` 的嵌套名字的访问与 `MyContainer` 的情形并无差别。在两种情形下，编译器只需检查类的内容便可确定 `ElemT` 是否为一个类型名字。

然而，一旦进入模板的上下文，事情就变得不同了，因为此时能够得到的精确信息不多。考虑如下代码片断：

```
template <typename T>
void aFuncTemplate( T &arg ) {
    ...T::ElemT...
```

当编译器遇到限定名 `T::ElemT` 时，它究竟知道些什么呢？从模板参数列表中，它知道 `T` 是某种类型名字。它还可以确定 `T` 是一个类的名字，因为我们使用了一个作用域操作符 `::` 来访问 `T` 的嵌套的名字。但是编译器知道的也就这么多了，因为其中没有任何关于 `T` 的内容的信息。举个例子，可以使用一个 `PtrList` 来调用 `aFuncTemplate`，此时 `T::ElemT` 为一个类型名字：

```
PtrList<State> states;
//...
aFuncTemplate( states ); // T::ElemT 是 PtrList<State>::ElemT
```

171

但是，如果采用一个不同的类型来实例化 `aFuncTemplate` 呢？

```
struct X {
    enum Types { typeA, typeB, typeC } ElemT;
    //...
};
X anX;
//...

aFuncTemplate( anX ); // T::ElemT 是 X::ElemT
```

在这种情况下，`T::ElemT` 是一个数据成员的名字，即为一个非类型的名字。那么编译器会怎么做呢？C++标准对此抱着尽力而为的态度，倘若编译器无法确定一个嵌套名字的类型，它就会假定嵌套的名字是一个非类型的名字。至此，导致 `fill` 函数模板出现语法错误的原因已经大白于天下！

为了对付这种情形，我们有时必须明确地通知编译器，某个嵌套的名字是一个类型名字：

```
template <typename T>
void aFuncTemplate( T &arg ) {
    ...typename T::ElemT...
```

在这里，使用 `typename` 关键字明确地告诉编译器，接下来的限定名字是一个类型名字，从而允许编译器去正确地解析模板。注意，我们是在告诉编译器 `ElemT` 是一个类型名字，而不是告诉它 `T` 是一个类型名字，因为编译器已经可以确定 `T` 是一个类型名字了。类似地，如果写

```
typename A::B::C::D::E
```

等于是在告诉编译器，嵌套层次最深的那个名字 `E` 是一个类型名字。

当然，如果拿一个不能满足被解析的模板要求的类型来实例化 `aFuncTemplate`，就会导致一个编译期错误：

```
struct Z {
    // 不存在一个名为 ElemT 的成员
};
Z aZ;
//...
aFuncTemplate( aZ ); // 错误！不存在 Z::ElemT 成员
aFuncTemplate( anX ); // 错误！X::ElemT 不是一个类型名字
aFuncTemplate( states ); // OK。嵌套的 ElemT 是一个类型名字
```

至此，我们可以改写 `fill` 函数模板，使其能够被正确地解析：

```
template <class Cont>
void fill( Cont &c, typename Cont::ElemT a[], int len ) { // OK
    for( int i = 0; i < len; ++i )
        c.insert( a[i] );
}
```

条款 50

成员模板

类模板的一些成员本身并不是模板，其中许多成员可以定义在类外。让我们看一个单链表容器：

```
template <typename T>
class SList {
public:
    SList() : head_(0) {}
    //...
    void push_front( const T &val );
    void pop_front();
    T front() const;
    void reverse();
    bool empty() const;
private:
    struct Node {
        Node *next_;
        T el_;
    };
    Node *head_; // -> list
};
```

当模板的成员函数定义在类模板外面时，它带有一个模板头部，其结构和用于类模板定义中的一样：

```
template <typename T>
bool SList<T>::empty() const
{ return head_ == 0; }
```

我们决定将这个单链表实现为指向一系列节点的指针，其中每一个节点包含有一个元素列表以及一个指向列表中下一个节点的指针（一个更为精密复杂的实现可能会嵌入一个平头的 Node (truncated Node)，而不是一个指向 Node 的指针，但对于我们这里的要求来说，目前的方式足够了）。一般来说，这样的嵌套类类型定义于模板自身内部，但未必非得如此：

```

template <typename T>
class SList {
public:
    //...
private:
    struct Node; // 不完整的类声明
    Node *head_; // -> list
    //...
};

template <typename T> // 定义于模板之外
struct SList<T>::Node {
    Node *next_;
    T el_;
};

```

成员 `empty` 和 `Node` 是模板成员的例子。但一个类模板（甚至一个非模板的类）还可以有成员模板（是的，我们又目击了一个例子，证明 C++ 喜欢定义容易混淆的技术术语。模板成员/成员模板、`new` 操作符/operator `new`、协变性/逆变性（*covariance/contravariance*）以及 `const_iterator`/常量迭代器等成对术语一起，使得每一次设计复审都像是在冒险）。遵照同义反复的传统，我们说，一个成员模板就是一个自身是模板的成员：

```

template <typename T>
class SList {
public:
    //...
    template <typename In> SList( In begin, In end );
    //...
};

```

和默认的构造函数不同，这个 `SList` 构造函数是一个成员模板，显式地采用类型名字 `In` 进行参数化。它同时还被用于实例化 `SList` 模板的类型名字隐式地参数化。这解释了为何当该构造函数定义于类模板之外时，它的定义看上去具有高度重复的特性：

```

template <typename T> // 这一个针对 SList
template <typename In> // 这一个针对成员
SList<T>::SList( In begin, In end ) : head_( 0 ) {
    while( begin != end )
        push_front( *begin++ );
    reverse();
}

```

就像对待其他函数模板一样，编译器会根据需要执行实参推导并实例化构造函数模板

(参见“模板实参推导[条款 57]”):

```
float rds[] = { ... };
const int size = sizeof(rds)/sizeof(rds[0]);
std::vector<double> rds2( rds, rds+size );
//...
SList<float> data( rds, rds+size ); // In是float *
SList<double> data2( rds2.begin(), rds2.end() ); // In是
//vector<double>::iterator
```

在 STL 中, 这是一个常见的构造函数模板应用, 以便允许一个容器可以通过从任一数据源中抽取的一系列值进行初始化。关于成员模板的另一个常见应用是生成类似复制操作的构造函数以及赋值操作符:

```
template <typename T>
class SList {
public:
    //...
    template <typename S>
        SList( const SList<S> &that );
    template <typename S>
        SList &operator =( const SList<S> &rhs );
    //...
};
```

175 这些模板成员可以用于类似复制构造函数和类似复制赋值的操作:

```
SList<double> data3( data ); // T是double, S是float
data = data3; // T是float, S是double
```

请注意上面描述的“类似复制构造函数”和“类似复制赋值”这两个无聊的短语。这是因为模板成员永远不会被用于实例化一个复制操作。确切地说, 如果上面的 T 和 S 是同样的类型, 那么编译器将不会实例化成员模板, 它自己将会“编写”一个复制操作。在这种情形下, 为了明确地防止编译器多管闲事 (往往是在帮倒忙), 通常最好明确地定义复制操作:

```
template <typename T>
class SList {
public:
    //...
    SList( const SList &that ); // 复制构造函数
    SList &operator =( const SList &rhs ); // 复制赋值操作符
    template <typename S> SList( const SList<S> &that );
    template <typename S>
```

```

        SList &operator =( const SList<S> &rhs );
        //...
    };
    //...
    SList<float> data4( data ); // 复制构造函数
    data3 = data2; // 复制赋值
    data3 = data4; // 成员模板支持的操作，并非复制赋值

```

任何非虚的成员函数都可以写成模板（成员模板不能是虚拟的，因为这些特性的组合会导致在其实现中存在难以克服的技术问题）。例如，可以为 SList 实现一个 sort 操作：

```

template <typename T>
class SList {
public:
    //...
    template <typename Comp> void sort( Comp comp );
    //...
};

```

这个 sort 成员模板允许用户传入一个可对列表中的元素进行比较的函数指针或函数对象（参见“STL 函数对象[条款 20]”）：

```

data.sort( std::less<float>() ); // 按升序 sort
data.sort( std::greater<float>() ); // 按降序 sort

```

在这里，分别采用标准函数对象类型 less 和 greater 实例化了两个不同版本的 sort 成员。

条款 51

采用 template 消除歧义

在条款“采用 typename 消除歧义[条款 49]”中，我们看到有时必须明确地告诉编译器，某个嵌套的名字是一个类型名字（而不是一个非类型名字），这样编译器才能正确地执行解析工作。对于嵌套的模板名字来说，存在同样的情况。

这方面典型的例子出现于 STL 配置器的实现中。如果你不熟悉 STL 配置器，也别闷闷不乐，因为对于理解接下来的讨论，并不需要预先熟悉 STL 配置器，当然了，足够的耐心还是需要的。

配置器是一种类类型，用于为 STL 容器定制内存管理操作。配置器通常采用类模板实现：

```
template <class T>
class AnAlloc {
public:
    //...
    template <class Other>
    class rebind {
    public:
        typedef AnAlloc<Other> other;
    };

    //...
};
```

类模板 AnAlloc 包含一个嵌套名字 rebind，后者自身也是一个类模板。它用于在 STL 框架内创建一个配置器，如同原先用于实例化一个容器的配置器一样，但它用于另一种不同的元素类型。例如：

```
typedef AnAlloc<int> AI; // 最初的配置器用于分配 int 元素
typedef AI::rebind<double>::other AD; // 分配 double 元素
typedef AnAlloc<double> AD; // 合法！这是一个相同的类型
```

这看上去也许有点古怪，然而使用这种 rebind 机制允许为不同的元素类型根据现有配

置器创建一个新版本的配置器，同时无需知道配置器的类型或者元素的类型：

```
typedef SomeAlloc::rebind<Node>::other NodeAlloc;
```

如果类型名字 `SomeAlloc` 遵从 STL 配置器的惯例，那么它将拥有一个嵌套的 `rebind` 类模板。本质上，等于在说，“我不知道这是哪种类型的配置器，也不知道它配置的是什么，但是我需要一个像配置 `Node` 那样的配置器！”

这种级别的“不知情”只能发生于模板的内部，在这种情形下，只有当模板被实例化时，才知道精确的类型和值。考虑一个针对 `SList` 容器（参见“成员模板[条款 50]”）的增强版本。这个增强版包含一个配置器类型 `A`，它可以配置 `T` 类型的元素。如同标准容器一样，`SList` 将会提供一个默认的配置器参数：

```
template < typename T, class A = std::allocator<T> >
class SList {
    //...
    struct Node {
        //...
    };
    typedef A::rebind<Node>::other NodeAlloc; // 语法错误！
    //...
};
```

就像列表以及其他基于节点的容器一样，元素类型为 `T` 的列表并不是真的配置和操纵类型为 `T` 的元素。实际上，它配置和操纵的是包含有一个类型为 `T` 的成员的节点。这正是上面所描述的情形。我们拥有某种配置器，它知道如何去配置类型为 `T` 的对象，但我们希望配置类型为 `Node` 的对象。然而，当试图去 `rebind` 时，却被告知语法错误。

再一次，问题出在此时编译器没有关于类型名字 `A` 的信息。按照 C++ 标准规定，编译器不得不假定嵌套的名字 `rebind` 是一个非模板的名字，并将紧随其后的尖括号解析为一个“小于”操作符。似乎问题已经很清楚了，不是吗？不，问题才刚刚开始。因为，即使编译器通过某种手段能够确定 `rebind` 是一个模板名字，但是，当它解析到另一个双重嵌套的名字 `other` 时，它又不得不假定那个名字是一个非类型的名字！到了给出解决方案的时候了。那个 `typedef` 必须被改写如下：

```
typedef typename A::template rebind<Node>::other NodeAlloc;
```

关键字 `template` 的使用等于告诉编译器，`rebind` 是一个模板名字，而 `typename` 的使用则等于告诉编译器，整个这一坨东西表示的是一个类型名字。很简单，不是吗？

180

181

条款 52

针对类型信息的特化

类模板显式特化和局部特化通常用于生成主类模板的一些版本，这些版本根据具体的模板实参或模板实参的类定制而成（参见“类模板显式特化[条款 46]”和“模板局部特化[条款 47]”）。

然而，这些语言特性常常也被以相反的样式使用，即，不是基于类型的属性生成特化版本，而是从一个特化版本中推导出类型的属性。让我们看一个简单的例子：

```
template <typename T>
struct IsInt // T不是一个int……
{ enum { result = false }; };
template <>
struct IsInt<int> // 除非T是一个int!
{ enum { result = true }; };
```

在继续讨论之前，首先要指出一点，一旦理解了那有点费解的语法，上面的代码其实是多么的简单。这是一个简单的众所周知的模板元编程的例子，确切地说，就是利用模板实例化机制，在编译期而非运行期执行一部分计算。这听起来挺玄乎的，但对于我那直言不讳的种植蔓越桔的邻居来说，仍然可以归结为一个简单的观察，“如果是它是一个 int 的话，那么它就是一个 int”。最高级的 C++ 模板编程并不比这个简单的例子困难多少，不过涉及的东西更多而已。

有了这个主模板和完全特化版本，就可以（在编译期）询问一个未知的类型是否为 int：

```
template <typename X>
void aFunc( X &arg ) {
    //...

    ...IsInt<X>::result...
    //...
}
```

在编译期向类型询问此类问题的能力，是许多重要的优化机制和错误检查技术的基础。当然，知道一个特定的类型是否恰好是一个 int，其作用很有限。但是，知道一个类型

是否为一个指针应该有更普遍的意义，因为一些实现往往根据所处理的类型到底是指向对象的指针还是对象本身而采取不同的形式：

```
struct Yes {}; // 一个对应于 true 的类型
struct No {}; // 一个对应于 false 的类型

template <typename T>
struct IsPtr // T 不是一个指针……
    { enum { result = false }; typedef No Result; };
template <typename T>
struct IsPtr<T*> // 除非它是一个不带修饰符的指针
    { enum { result = true }; typedef Yes Result; };
template <typename T>
struct IsPtr<T*const> // 或者是一个 const 指针
    { enum { result = true }; typedef Yes Result; };
template <typename T>
struct IsPtr<T*volatile> // 或者是一个 volatile 指针
    { enum { result = true }; typedef Yes Result; };
template <typename T>
struct IsPtr<T*const volatile> // 或者是一个 const volatile 指针
    { enum { result = true }; typedef Yes Result; };
```

和 IsInt 的情形相比，我们询问了 IsPtr 更一般性的问题。我们是在利用局部特化机制来“捕获”带有形形色色指针修饰符的限定版本。正如上面提到的那样，这个 IsPtr 设施理解起来并不比 IsInt 设施困难多少，不过是有更多的语法上的挑战而已（参见“SFINAE [条款 59]”，以便了解类似的元编程技术）。

184

为了体会这种能力的作用（即可以在编译期询问某个类型一些问题），考虑如下简单的 Stack 模板实现：

```
template <typename T>
class Stack {
public:
    ~Stack();
    void push( const T &val );
    T &top();
    void pop();
    bool empty() const;
private:
    //...
    typedef std::deque<T> C;
    typedef typename C::iterator I;
    C s_;
};
```

这个 Stack 不过是对标准 deque 进行包装的一个友好接口而已，它类似于标准 stack 容器适配器 (container adapter)。其中大多数操作都很直观，可以直接采用 deque 来实现：

```
template <typename T>
void Stack<T>::push( const T &val )
    { s_.push_back( val ); }
```

然而，Stack 的析构函数有个问题。当一个 Stack 对象被销毁时，它的 deque 数据成员同样将被销毁，从而又导致销毁 deque 中包含的任何元素。如果这些元素是指针的话，那么，它们所指向的对象将不会被销毁。标准容器 deque 的行为就是这样的。因而，必须为 Stack 决定一个指针元素删除策略，对于这样的情形只要声明 delete 即可（另请参考“policy [条款 56]”，以便了解一个更有弹性的方式）。然而，我们并不能简单地叫析构函数去 delete deque 的元素，因为如果元素类型并不是指针的话，这种做法就是错误的。

185

解决方案之一是使用 Stack（主）模板的局部特化来处理元素类型为指针的栈（参见“模板局部特化[条款 47]”）。然而，由于 Stack 只有一小部分的行为需要做出改变，这种方式看上去“动作过大”。另一种方式是仅仅（在编译期）询问明显的问题并作出相应的反应：“如果 Stack 的元素类型是指针，那么就 delete 其中包含的任何元素，否则不要去 delete。”

```
template <typename T>
class Stack {
public:
    ~Stack()
        { cleanup( typename IsPtr<T>::Result() ); }
    //...
private:
    void cleanup( Yes ) {
        for( I i( s_.begin() ); i != s_.end(); ++i )
            delete *i;
    }
    void cleanup( No )
        {}
    typedef std::deque<T> C;
    typedef typename C::iterator I;
    C s_;
};
```

其中有两个不同的 cleanup 成员函数，一个带有一个类型为 Yes 的参数，另一个带有一个类型为 No 的参数。Yes 版本的 cleanup 执行 delete 操作，No 版本则不执行。

析构函数利用 `T` 去实例化 `IsPtr` 并访问其嵌套的类型名字 `Result` 来询问：“`T` 是一个指针类型吗？”（参见“采用 `typename` 消除歧义[条款 49]”），答案要么是 `Yes`，要么是 `No`，并将该类型的一个对象传递给 `cleanup` 函数。请注意，只有其中一个版本的 `cleanup` 将会被实例化并被调用，另一个则不会（参见“只实例化要用的东西[条款 61]”）。

```
Stack<Shape *> shapes; // 将会 delete
Stack<std::string> names; // 将不会 delete
```

186

类模板特化可被用于从类型中提取任意复杂的信息。例如，我们可能不但想知道一个特定的类型是否是一个数组，还想知道，如果它确实是一个数组，那么它是一个什么类型的数组，其边界值是多少：

```
template <typename T>
struct IsArray { // T 不是一个数组……
    enum { result = false };
    typedef No Result;
};

template <typename E, int b>
struct IsArray<E [b]> { // 除非它是一个数组！
    enum { result = true };
    typedef Yes Result;
    enum { bound = b }; // 数组边界
    typedef E Etype; // 数组元素类型
};
```

我们可能不但想知道一个特定的类型是否是一个指向数据成员的指针，还想知道如果它确实是的话，类的类型是什么，成员类型又是什么：

```
template <typename T>
struct IsPCM { // T 不是一个指向数据成员的指针
    enum { result = false };
    typedef No Result;
};

template <class C, typename T>
struct IsPCM<T C::*> { // 除非它是！
    enum { result = true };
    typedef Yes Result;
    typedef C ClassType; // 类的类型
    typedef T MemberType; // 类成员的类型
};
```

有许多流行的工具包利用了这些技术，藉此提供访问类型特征的能力（参见“`traits` [条款 54]”），用于在编译期执行代码定制工作。

187

条款 53

嵌入的类型信息

我们怎么才能知道一个容器中的元素的类型呢？

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

乍看上去这似乎根本不是个问题。Seq<std::string>的元素类型是std::string，不是吗？没错，但事情并非总是如此简单。没有任何东西可以阻止那些（非标准）序列容器的实现采用 const T、T *或针对 T 的“智能指针”作为元素的类型（一个尤其怪异的容器实现说不定会忽视模板参数的确切类型并总是将元素的类型设置为 void *!）。此外，这些行为怪异的实现并不是我们希望能够决定容器元素类型的惟一原因，我们还常常在缺乏这种信息的情况下编写泛型代码：

```
template <class Container>
Elem process( Container &c, int size ) {
    Temp temp = Elem();
    for( int i = 0; i < size; ++i )
        temp += c[i];
    return temp;
}
```

在上面的泛型算法 process 中，我们需要知道 Container 的元素类型 (Elem)，以及一个可以用作声明一个临时对象（用于持有元素类型的对象）的类型 (Temp)，但只有到 process 函数模板被具体的容器进行实例化时，该信息才可用。

处理这种情形的一种常见的方式是叫类型自己提供“个人”信息。这种信息通常嵌入于类型自身，就像嵌入在人体内的一个微型芯片那样，可以用于查询人的名字、身份证号、血型等等（这只是一个类比而已，并不代表我赞成对人类干这种事）。我们对容器的血型没什么兴趣，但对其元素的类型兴趣还是蛮大的：

```
template <class T>
class Seq {
```

```

public:
    typedef T Elem; // 元素的类型
    typedef T Temp; // 临时对象的类型
    size_t size() const;
    //...
};

```

在编译期可以查询到这种嵌入的信息：

```

typedef Seq<std::string> Strings;
//...
Strings::Elem aString;

```

这种方式对于使用标准库容器的任何用户来说都不会陌生。例如，为了给一个标准容器声明一个迭代器，明智的做法是询问容器其迭代器类型是什么：

```

vector<int> aVec;
//...
for( vector<int>::iterator i( aVec.begin() );
    i != aVec.end(); ++i )
    //...

```

这里我们通过询问`vector<int>`来了解它的迭代器类型是什么，而不是假定它是`int *`（尽管对于许多实现来说通常的确如此）。`vector<int>`的迭代器类型也可以是其他什么类型（比如用户自定义的安全的指针类型），因此最具移植性的循环编写方式是从`vector<int>`自身获得迭代器类型。

一个更重要的观察是，这种方式允许我们假定所需的信息已经被提供了，从而可以编写泛型代码：

```

template <class Container>
typename Container::Elem process( Container &c, int size ) {
    typename Container::Temp temp
        = typename Container::Elem();
    for( int i = 0; i < size; ++i )
        temp += c[i];
    return temp;
}

```

190

这个版本的`process`算法查询`Container`类型的个人信息，并假定`Container`定义了嵌套类型名字`Elem`和`Temp`（注意，我们必须在三个地方使用`typename`关键字来明确地告诉编译器，嵌套的名字是类型名，而不是其他什么嵌套名字。参见“采用`typename`消除歧义[条款 49]”）。

```
Strings strings;  
aString = process( strings, strings.size() ); // OK
```

这个 process 算法可以与 Seq 容器很好地协作，并且也能够与任何遵从约定的其他任何容器协作：

```
template <typename T>  
class ReadonlySeq {  
public:  
    typedef const T Elem;  
    typedef T Temp;  
    //...  
};
```

191 我们可以处理一个 ReadonlySeq 容器，因为它符合我们的约定。

条款 54

traits^①

有时仅仅知道对象的类型是不够的，通常来说，有些与对象类型有关的信息对于处理对象来说是不可或缺的。在“嵌入的类型信息[条款 53]”中，我们看到了诸如标准容器这样的复杂类型通常是如何在其内部嵌入与自身有关的东西的：

```
Strings strings;
aString = process( strings, strings.size() ); // OK
std::vector<std::string> strings2;
aString = process( strings2, strings2.size() ); // 错误!
extern double readings[RSIZ];
double r = process( readings, RSIZ ); // 错误!
```

process 算法与 Seq 容器能够很好地协作，但面对标准容器 vector 却失败了，因为 vector 并没有定义（process 假定存在的）嵌套的类型名字。

我们可以 process 一个 ReadonlySeq，因为它符合我们的假定，但可能还想去 process 那些不遵从我们非常狭小约定的容器，并且也可能希望去 process 类似容器一般的东西——这个东西甚至可以不是一个类。traits 类通常用于解决这个问题。

traits 类是一个关于某个类型的信息的集合。然而，与嵌套的容器信息不同的是，traits 类独立于它所描述的类型。

```
template <typename Cont>
struct ContainerTraits;
```

有关 traits 类的一个常见的应用，是在泛型算法和不遵从算法期望的约定的类型之间，放入一个遵从约定的中间层。根据类型的 traits 编写算法。在一般情况下通常假设有某种约定。在这个例子中，ContainerTraits 将对 Seq 和 ReadonlySeq 容器所使用的约定作出

① traits：特征、特性等。明显起见，本条款基本保留不译。另外，本条款提到的 traits class（特征类、特征萃取类），实际上是 traits class template，这种类模板的成员用于描述模板实参的特征。除了本条款所述作用外，traits 往往还用于避免（其他）模板参数数目过多。——译者注

193 假设。

```
template <typename Cont>
struct ContainerTraits {
    typedef typename Cont::Elem Elem;
    typedef typename Cont::Temp Temp;
    typedef typename Cont::Ptr Ptr;
};
```

有了这附加的 traits 类模板，就可以选择引用容器类型中嵌套的 Elem 类型，通过容器类型，或通过使用容器类型实例化的 traits 类型，都可以做到这一点。

```
typedef Seq<int> Cont;
Cont::Elem e1;
ContainerTraits<Cont>::Elem e2; // 与 e1 的类型相同
```

可以改写 process 泛型算法，利用 traits 来代替对容器嵌套类型名字的直接访问：

```
template <typename Container>
typename ContainerTraits<Container>::Elem
process( Container &c, int size ) {
    typename ContainerTraits<Container>::Temp temp
        = typename ContainerTraits<Container>::Elem();
    for( int i = 0; i < size; ++i )
        temp += c[i];
    return temp;
}
```

看上去我们似乎在自找麻烦，把泛型算法 process 的语法弄得更加让人费解了！先前，为了获得容器中的元素的类型，我们写 `typename Container::Elem`。我们以直白的语言说，“获得 Container 的嵌套名字 Elem，顺带一提，它是一个类型名字”。而使用 traits 时就必须写 `typename ContainerTraits<Container>::Elem`。本质上，等于是在说，“请实例化对应于 Container 的 ContainerTraits 类，并且获取其嵌套的名字 Elem，顺带一提，它是一个类型名字”。我们往后退了一步，即不是直接从容器类型自身获得信息，而是从中间层 traits 类获取信息。如果将访问嵌套的类型信息比喻为从人体内的一个嵌入芯片中读取个人信息，那么使用 traits 类就像使用人的名字作为键值到一个数据库中查找此人的信息。你将会得到和直接查找方式所得的相同的信息，但数据库查找方式无疑不是那么具有侵入性，而且更灵活。

194

比方说，如果一个人没有安装芯片，就不能从他那里获得信息（也许此人来自一个将嵌入芯片视作不合乎风俗的地方）。然而，总是可以在数据库中为这样的人创建一个新的条目，甚至无需通知有关个体。类似地，可以对 traits 模板进行特化，以便提供关于特定

的非兼容容器的信息，而且不会对容器自身造成任何影响：

```
class ForeignContainer {
    // 不带嵌入的类型信息……
};
//...
template <>
struct ContainerTraits<ForeignContainer> {
    typedef int Elem;
    typedef Elem Temp;
    typedef Elem *Ptr;
};
```

有了这个ContainerTraits特化，就可以有效地process一个ForeignContainer，就像按照我们的约定所编写的那样。原来的process实现面对ForeignContainer将会失败，因为它试图去访问一个不存在的嵌套信息：

```
ForeignContainer::Elem x; // 错误，不存在这样的嵌套名字！
ContainerTraits<ForeignContainer>::Elem y; // OK，使用traits
```

将 traits 模板想象为一个信息集合，且该信息集合通过一个类型进行索引，这种看法比较有积极的意义，就像关联式容器通过关键字（key）进行索引一样。不过，对 traits 的“索引”发生于编译期（而非运行期），通过模板特化达成。

通过 traits 类来访问一个类型的信息的另一个优势在于，这项技术可被用于为不是类的类型（因而可以没有嵌套的信息）提供信息。纵然 traits 自身是类，但封装了（traits）的类型未必一定是类。例如，数组可以说是一种退化的容器（不管是从数学的意义上看，还是从实际的意义上看），我们希望能够像操纵正儿八经的容器那样来操纵它：

195

```
template <>
struct ContainerTraits<const char *> {
    typedef const char Elem;
    typedef char Temp;
    typedef const char *Ptr;
};
```

有了这个针对“容器”类型的 const char *特化，就可以 process 一个字符数组：

```
const char *name = "Arsene Lupin";
const char *r = process( name, strlen(name) );
```

我们可以对其他类型的数组照葫芦画瓢，一一为 int *、const double * 等生成特

化。不过，定义可以处理任何指针类型的单个版本会更方便，因为不同的指针类型具有相似的属性。为了达到这个目的，可以利用 traits 模板针对指针的局部特化版本：

```
template <typename T>
struct ContainerTraits<T *> {
    typedef T Elem;
    typedef T Temp;
    typedef T *Ptr;
};
```

采用任何指针类型来特化 ContainerTraits——不管是 `int *` 还是 `const float *` (`*const*`)(`int`)，都会导致对这个局部特化版本的实例化，除非存在一个更加特化的 ContainerTraits 版本：

```
extern double readings[RSIZ];
double r = process( readings, RSIZ ); // 可以工作！
```

然而，事情还没完。注意，对指向常量的指针来使用这个局部特化版，将不会产生用于临时对象 (Temp) 的正确类型。确切地说，常量临时对象值没什么用处，因为无法赋值给它们。我们希望得到类似元素类型的非常量值来作为临时对象的类型。举个例子，对于 `const char *` 的情况而言，`ContainerTraits<const char *>::Temp` 应该为 `char`，而不应该是 `const char`。可以利用一个额外的局部特化来处理这种情形：

```
template <typename T>
struct ContainerTraits<const T *> {
    typedef const T Elem;
    typedef T Temp; // 注意：类似 Elem 的非常量值
    typedef const T *Ptr;
};
```

当模板实参为一个指向常量的指针（而不是一个指向非常量的指针）时，这个更具针对性的局部特化将会优先于前一个局部特化而被选择。

局部特化还可以帮助我们将 traits 机制扩展至下面这种用途：将一个“外部”的约定转换为符合本地的约定。例如，STL 对约定有很强的依赖（参见“STL[条款 4]”）。标准容器具有类似于封装在 ContainerTraits 中的概念，但表达的方式不同。比如，我们先曾尝试使用标准 vector 来实例化 process 算法，但失败了，现在让我们来摆平这件事：

```
template <class T>
struct ContainerTraits< std::vector<T> > {
    typedef typename std::vector<T>::value_type Elem;
```

```
typedef typename
    std::iterator_traits<typename
    std::vector<T>::iterator>
    ::value_type Temp;
typedef typename
    std::iterator_traits<typename
    std::vector<T>::iterator>
    ::pointer Ptr;
};
```

这并不是我们能够想到的最具可读性的实现，不过还好，对于用户来说，它是隐藏的，用户现在可以利用从标准 vector 生成的容器来调用泛型算法：

```
std::vector<std::string> strings2;
aString = process( strings2, strings2.size() ); // 可以工作了!
```

条款 55

模板的模板参数

让我们重新拾起在条款“针对类型信息的特化[条款 52]”中讨论过的 Stack 模板。我们决定使用一个标准 deque 来实现它，这是一个相当好的折衷实现方式，尽管在许多场合使用别的容器会更高效或更合适。可以通过向 Stack 添加额外的模板参数来解决这个问题，该参数表示实现所使用的容器类型：

```
template <typename T, class Cont>
class Stack;
```

为了简化，先放弃使用标准库（其实，这通常不是好主意），并假定已经有一些非标准容器模板可供使用，包括 List、Vector、Deque，等等。让我们还假定这些容器类似于标准容器，但它们只具有单个模板参数，该参数表示容器的元素类型。

让我们回忆一下。标准容器实际上至少有两个参数：一个表示元素类型，另一个表示配置器类型。容器使用配置器来分配和释放工作内存，从而使得这种行为可被按需定制。从效果上说，配置器为容器指定内存管理策略（参见“policy [条款 56]”）。因为容器存在缺省的配置器，因此我们很容易忘记有这么一回事。实际上，当实例化一个像 `vector<int>` 这样的标准容器时，得到的其实是一个 `vector<int, std::allocator<int>>`。

例如，非标准 List 的声明为：

```
template <typename> class List;
```

可以注意到，在 List 的声明中模板参数的名字被省略了。如同函数声明中的形参名字一样，在模板声明中是否赋予模板参数一个名字也是可选的。而且如同函数定义一样，只有在模板定义中并且仅当参数名字被模板定义所使用时，模板参数的名字才是必需的。然而，和函数声明中的形参一样，在模板声明中给模板参数起个名字对于该模板的文档化还是很有帮助的：

```
template <typename T, class Cont>
class Stack {
public:
    ~Stack();
    void push( const T & );
```

```

    //...
private:
    Cont s_;
};

```

Stack 的用户现在可以提供两个模板实参，一个表示元素的类型，另一个表示容器的类型，并且容器必须能够容纳该元素类型的对象：

```

Stack<int, List<int> > aStack1; // OK
Stack<double, List<int> > aStack2; // 合法，但有问题
Stack<std::string, Deque<char *> > aStack3; // 错误!

```

aStack2 和 aStack3 的声明展示了可能存在的参数协调一致的问题。如果用户为元素类型选择了错误的容器类型，将会得到一个编译期错误（在 aStack3 的情形下，因为不能将一个 string 复制给一个 char *），或者一个微妙的 bug（在 aStack2 的情形下，因为从 double 到 int 的复制会丢失精度）。此外，大多数 Stack 用户不想为选择底层实现而烦神，因此提供一个合情合理的缺省实现应该是令人满意的。可以通过为第二个模板参数提供一个缺省值而改善这种情形：

```

template <typename T, class Cont = Deque<T> >
class Stack {
    //...
};

```

如果 Stack 的用户乐于接受一个 Deque 实现或者不是特别关心其实现时，这很有助益：

```

Stack<int> aStack1; // 容器是 Deque<int>
Stack<double> aStack2; // 容器是 Deque<double>

```

200

这多少也是标准容器适配器 stack、queue 以及 priority_queue 所采用的方式：

```

std::stack<int> stds; // 容器为 deque< int, allocator<int> >

```

这种方式是对方便性和灵活性的一个良好的折衷。对于那些偶尔尝试 Stack 设施的用户来说，这种方式提供了方便；而对于那些希望利用任何（合法且有效的）容器来容纳 Stack 元素的有经验的用户来说，这种方式则提供了足够的灵活性。

然而，这种灵活性是以安全性为代价的。因为当使用其他容器进行特化时，这种方式仍然需要协调元素和容器的类型，显然，这种对类型协调的要求，就可能会带来不协调：

```

Stack<int, List<int> > aStack3;
Stack<int, List<unsigned> > aStack4; // 哎呀!

```

让我们来看看能否在提高安全性的同时保持说得过去的灵活性。一个模板可以带有一个

自身就是一个模板名字的参数。这种参数具有让人高兴的重复名字：模板的模板参数：

201

```
template <typename T, template <typename> class Cont>
class Stack;
```

这个新版的 Stack 模板参数列表看上去挺打击人信心的，其实不然，它实际上并不像看上去的那样糟糕。我给大家解释一下，第一个参数 T 还是老一套，不过是一个类型的名字而已。第二个参数 Cont 是一个模板的模板参数。它是一个类模板的名字，该类模板带有单个类型名字参数。注意，我们没有给出 Cont 的类型名字参数的名字，当然，我们完全可以给出一个：

```
template <typename T, template <typename ElementType> class Cont>
class Stack;
```

然而，这样的名字（ElementType，如上）不过是充任文档之用而已，如同函数声明中形参的名字一样。这些名字通常被省略，但是，如果认为它们的存在可以改善可读性，那儿尽管写上它们好了。反过来说，如果将 Stack 声明中技术上不需要的名字都消除了，就会将其可读性降至最低：

```
template <typename, template <typename> class>
class Stack;
```

但是，出于对代码读者的同情，这样的编程实践应该严加限制，尽管 C++ 语言没有强制规定我们不要这么做。

Stack 模板使用其类型名字参数来实例化其模板的模板参数，所得到的容器类型用于实现 Stack：

```
template <typename T, template <typename> class Cont>
class Stack {
    //...
private:
    Cont<T> s_;
};
```

这种方式允许元素和容器类型之间的协调问题通过 Stack 自身的实现来解决，而不是在对 Stack 进行特化的各种不同的代码中进行解决。这种“单点特化”方式大大降低了元素类型和用于容纳该种元素的容器类型之间不协调的可能性：

```
Stack<int, List> aStack1;
Stack<std::string, Deque> aStack2;
```

为了得到一个额外的便利，可以为模板的模板参数指定一个缺省值：

```
template <typename T, template <typename> class Cont = Deque>
class Stack {
    //...
};
//...
Stack<int> aStack1; // 使用缺省的参数值, 即 Cont 是一个 Deque
Stack<std::string, List> aStack2; // Cont 是一个 List
```

对于处理模板的一套实参和一个采用这套实参进行实例化的模板之间的协调问题, 以上方式通常很棒。

将模板的模板参数与恰好是从模板生成的类型名字参数混淆, 是很常见的事情。例如, 考虑如下类模板声明:

```
template <class Cont> class Wrapper1;
```

202

模板 Wrapper1 需要一个类型名字作为其模板实参 (在 Wrapper1 的 Cont 参数的声明中, 使用关键字 class 而不是 typename, 目的是为了告诉代码的读者, 期望的是一个 class 或 struct, 而不是任意一个类型。不过这对于编译器来说没有什么区别。在此上下文中, 从技术上来说, typename 与 class 完全是相同的东西。参见“可选的关键字[条款 63]”)。这个类型名字当然可以从一个模板生成, 例如 Wrapper1< List<int> > 中的实参就是如此, 但是 List<int> 不过是一个类的名字, 尽管它产生于一个模板:

```
Wrapper1< List<int> > w1; // 很好, List<int> 是一个类型名字
Wrapper1< std::list<int> > w2; // 很好, list<int> 是一个类型名字
Wrapper1<List> w3; // 错误! List 是一个模板名字
```

作为一种替代方式, 考虑如下的类模板声明:

```
template <template <typename> class Cont> class Wrapper2;
```

模板 Wrapper2 需要一个模板名字用作其模板实参, 但不是随便一个模板名字就可以的。此声明等于是说, 所需模板必须带有单个类型参数:

```
Wrapper2<List> w4; // 很好, List 是一个单参模板
Wrapper2< List<int> > w5; // 错误! List<int> 不是一个模板
Wrapper2<std::list> w6; // 错误! std::list 带有不止一个参数
```

如果希望能够采用标准容器进行特化, 必须这么做:

```
template <template <typename Element,
    class Allocator> class Cont>
class Wrapper3;
```

或采用以下等价的方式:

```
template <template <typename,typename> class Cont>  
class Wrapper3;
```

203 这个声明是说，模板必须带有两个类型名字参数：

```
Wrapper3<std::list> w7; // 也许可以工作……
```

```
Wrapper3< std::list<int> > w8; // 错误! list<int>是一个类
```

```
Wrapper3<List> w9; // 错误! List 只带有一个类型参数
```

值得一提的是，标准容器模板（比如 `list`）可能被声明为带有不止两个参数（这是合法的），因此上面 `w7` 的声明也许并不能在所有平台上工作。虽然我们都热爱并尊重 STL，但我们从来没有宣称它完美无瑕。

204

条款 56

policy^①

在条款“针对类型信息的特化[条款 52]”中，我们设计了一个栈（stack）模板，当栈的生命期结束时，如果栈中元素类型为指针，它可以 delete 栈中任何剩余的元素：

```
template <typename T> class Stack;
```

这种策略并非不合理，只是不够灵活。完全有可能栈的用户不希望 delete 栈中指针所指向的东西。比如说，指针所指的对象可能并不位于堆上，或者这些对象也被其他容器所共享。此外，很可能指针指向的是一个对象数组，而不是单个对象。以字符指针为元素的栈就属于这样的情形，字符指针通常指向一个以 null 结尾的字符串：

```
Stack<const char *> names; // 哎呀！未定义行为
```

删除策略假定 Stack 中的指针指向单个对象，因此将会使用非数组形式的 delete，其实对于数组来说，应该使用 array delete（参见“数组分配[条款 37]”）。

我们的目标是能够以类似如下的方式编写 Stack 模板的析构函数：

```
template <typename T>
class Stack {
public:
    ~Stack() {
        for( I i( s_.begin() ); i != s_.end(); ++i )
            doDeletionPolicy( *i );
    }
    //...
private:
    typedef std::deque<T> C;
    typedef typename C::iterator I;
    C s_;
};
```

① policy：策略。明显起见，本条款中基本保留不译。policy 是一个类或一个类模板，其成员用于描述泛型组件的可配置的行为。它通常用作（其他）模板的实参。——译者注

该析构函数遍历每一个剩下的元素并对它们执行适当的 **deletion policy**（删除策略）。**doDeletionPolicy**的实现可以是多种多样的。最典型的方式是，当 **Stack** 模板被实例化并采用模板的模板参数实现时（参见“模板的模板参数[条款 55]”），**policy** 就变得明确了：

```
template <typename T, template <typename> class DeletionPolicy>
class Stack {
public:
    ~Stack() {
        for( I i( s_.begin() ); i != s_.end(); ++i )
            DeletionPolicy<T>::doDelete( *i ); // 执行 policy
    }
    //...
private:
    typedef std::deque<T> C;
    typedef typename C::iterator I;
    C s_;
};
```

通过考查 **deletion policy** 在 **Stack** 析构函数中的应用，可以确定 **Stack** 中的 **DeletionPolicy** 是一个类模板，并且是采用 **Stack** 的元素类型进行实例化。它具有一个名为 **doDelete** 的静态成员函数，该函数负责对 **Stack** 的元素执行适当的删除动作。现在我们可以着手定义一些适当的策略，其中之一针对 **delete**：

```
template <typename T>
struct PtrDeletePolicy {
    static void doDelete( T ptr )
        { delete ptr; }
};
```

当然，也可以使用不同的接口来实现这个 **policy**。例如，我们可以重载函数调用操作符，而不是使用一个静态成员函数：

```
template <typename T>
struct PtrDeletePolicy {

    void operator ()( T ptr )
        { delete ptr; }
};
```

并且将 **Stack** 析构函数中的删除操作修改如下：

```
DeletionPolicy<T>()(*i);
```

在这里，建立一个众人遵守的约定非常重要，因为访问特定 `policy` 的每一个实现都将采用同样的语法。

其他一些有意义的策略执行数组删除操作或者什么事情都不做：

```
template <typename T>
struct ArrayDeletePolicy {
    static void doDelete( T ptr )
    { delete [] ptr; }
};

template <typename T>
struct NoDeletePolicy {
    static void doDelete( const T & )
    {}
};
```

现在，当实例化 `Stack` 时，就可以指定一个适当的 `deletion policy`：

```
Stack<int, NoDeletePolicy> s1; // 不要delete int 元素
Stack<std::string *, PtrDeletePolicy> s2; // delete string *
Stack<const char *, ArrayDeletePolicy> s3; // 执行delete []
Stack<const char *, NoDeletePolicy> s4; // 不要delete!
Stack<int, PtrDeletePolicy> s5; // 错误! 不可delete int!
```

如果某一个 `policy` 比其他策略更常用，通常最好把它作为缺省 `policy`：

```
template <typename T,
    template <typename> class DeletionPolicy = NoDeletePolicy>
class Stack;
//...
Stack<int> s6; // 不执行delete 操作
Stack<const char *> s7; // 不执行delete 操作
Stack<const char *, ArrayDeletePolicy> s8; // 执行delete []
```

207

在模板设计中，通常需要为用户提供一些通过策略进行参数化的机会。例如，在“模板的模板参数[条款55]”中，让用户能够指定如何实现一个 `Stack`。这就是一个实现策略 (implementation policy)：

```
template <typename T,
    template <typename> class DeletionPolicy = NoDeletePolicy
```

```
template <typename> class Cont = Deque>
class Stack;
```

这种做法为 Stack 用户带来了额外的灵活性：

```
Stack<double *, ArrayDeletePolicy, Vector> dailyReadings;
```

同时还允许在默认情况下提供良好的一般行为：

```
Stack<double> moreReadings; // 不执行 delete 操作，并且底层采用 Deque 实现
```

在泛型程序设计中，常常要做出关于实现和行为的策略决策。通常而言，这些决策可以
208 采用上文所谈的策略技术进行抽象和表示。

条款 57

模板实参推导

类模板必须被显式地予以特化。例如，如果希望对“类模板显式特化[条款 46]”中讨论的 `Heap` 容器进行特化，就必须为模板提供一个类型名字实参：

```
Heap<int> aHeap;
Heap<const char *> anotherHeap;
```

函数模板也可以被显式地特化。设想有一个函数模板，它执行一个受限制的旧式转型操作：

```
template <typename R, typename E>
R cast( const E &expr ) {
    // 做一些聪明的检查工作……
    return R( expr ); // 转型
}
```

当调用函数模板时，“可以”显式地特化该模板，就像“必须”特化一个类模板那样：

```
int a = cast<int,double>(12.3);
```

然而，典型的做法（同时也是更方便的做法）是让编译器根据函数调用中的实参类型推导出模板实参。这在情理之中，这个过程称为模板实参推导（**template argument deduction**）。不过要小心，在下文的描述中，请注意术语“模板实参”和“函数实参”之间的区别（参见“模板术语[条款 45]”）。让我们考虑一个模板，它带有单个模板实参，两个函数参数，此模板用于找出两个函数参数中较小的那一个：

```
template <typename T>
T min( const T &a, const T &b )
{ return a < b ? a : b; }
```

209

在未明确提供模板实参的情况下使用 `min` 时，编译器将会检查函数调用实参的类型，目的在于推导出模板实参：

```
int a = min( 12, 13 ); // T是int
double d = min( '\b', '\a' ); // T是char
char c = min( 12.3, 4 ); // 错误! T不能既是double又是int
```

上面错误的那一行代码，原因在于编译器无法在模棱两可的情形下推导出模板实参。在这种情形下，我们总是可以用显式的方式告诉编译器，模板实参究竟是什么：

```
d = min<double>( 12.3, 4 ); // OK, T是double
```

试图利用模板实参推导机制来使用 `cast` 模板时，也会发生类似的情形：

```
int a = cast( 12.3 ); // 错误! E是double，但R呢？
```

与重载解决方案一样，编译器在模板实参推导期间只检查函数实参的类型，对可能的返回类型并不作检查。因此，要想让编译器知道返回类型，除了明确地告诉它，别无他法：

```
int a = cast<int>( 12.3 ); // E是double，R被明确地告知是int
```

注意，如果编译器可以自行推导出它们，那么模板实参列表中靠后的模板实参可以省略。在这个例子中，只需向编译器提供返回类型（即 `R`）即可，编译器可以自行推导出表达式类型。对于模板的可用性而言，模板参数的顺序占着举足轻重的地位。就拿此例来说，如果表达式类型放置于返回类型之前，就不得不对二者都显式地予以指定。

行文至此，有不少人会注意到上面对 `cast` 的调用语法，并会问，“你莫非是在暗示 `static_cast`、`dynamic_cast`、`const_cast` 以及 `reinterpret_cast` 都是函数模板？”不，我没有这个意思，因为这四个转型操作符都不是模板，它们是内建的操作符（如同 `new` 操作符或针对整数的 `+` 操作符一样）。但几乎可以肯定的是，它们的语法受到了像 `cast` 函数模板这样的东西的启发（参见“新式转型操作符[条款 9]”）。

210

注意，模板实参推导是通过检查传递给调用中的实参类型而进行的。这就意味着任何无法从实参类型推导出来的模板实参都必须显式地予以提供。下面是一个不讨人喜欢的 `repeat` 函数模板：

```
template <int n, typename T>
void repeat( const T &msg ) {
    for( int i = 0; i < n; ++i )
        std::cout << msg << std::flush;
}
```

在这里，我们小心翼翼地将 `int` 模板实参放在类型实参 `T` 之前，这样仅仅指定消息（`message`）的重复次数就可以了，因为模板实参推导机制可以帮我们确定消息的类型：

```
repeat<12>( 42 ); // n是12, T是int
repeat<MAXINT>( '\a' ); // n是MAXINT, T是char
```

在 `cast`、`min` 和 `repeat` 模板中，编译器可以从单个函数实参中推导出单个模板实参，其实这种推导机制也能够从单个函数实参的类型推导出多个模板实参：

```
template <int bound, typename T>
void zeroOut( T (&ary)[bound] ) {
    for( int i = 0; i < bound; ++i )
        ary[i] = T();
}
//...
const int hrsinweek = 7*24;
float readings[hrsinweek];
zeroOut( readings ); // bound == 168, T是float
```

在上例中，`zeroOut` 期望一个数组实参，模板实参推导机制能够剖析该实参的类型，来确定数组的边界值和元素的类型。

我们注意到在本条款一开始说过，类模板必须被显式地特化。然而，函数模板实参推导机制可用于间接地特化类模板。考虑一个可用于从函数指针来生成函数对象的类模板（参见“STL 函数对象[条款 20]”）：

211

```
template <typename A1, typename A2, typename R>
class PFun2 : public std::binary_function<A1,A2,R> {
public:
    explicit PFun2( R (*fp)(A1,A2) ) : fp_( fp ) {}
    R operator()( A1 a1, A2 a2 ) const
        { return fp_( a1, a2 ); }
private:
    R (*fp_)(A1,A2);
};
```

（这是标准库中 `pointer_to_binary_function` 模板的简化版，我特意选定这个例子来说明对它进行特化的语法是多么地令人费解，其实它并不比这里看到的東西更糟糕）直接实例化这个模板有点麻烦：

```
bool isGreater( int, int );
std::sort(b, e, PFun2<int,int,bool>(isGreater)); // 痛苦！
```

对于此类情况，通常可以提供一個“辅助函数（helper function）”，该函数惟一的用途在于推导模板实参，为的是可以像施展魔法一般去特化一个类模板：

```
template <typename R, typename A1, typename A2>
inline PFun2<A1,A2,R> makePFun( R (*pf)(A1,A2) )
    { return PFun2<A1,A2,R>(pf); }
//...
std::sort(b, e, makePFun(isGreater)); // 好多了
```

在这个推导特技中，编译器能够从单个函数实参的类型推导出实参类型和返回值类型。这项技术被广泛应用于标准库中的 `ptr_fun`、`make_pair`、`mem_fun`、`back_inserter` 以及其他一些辅助函数中，以便更容易完成那些复杂的、易于出错的类模板特化任务。

条款 58

重载函数模板

一个函数模板可以被其他函数模板和非模板函数重载。这种能力非常有用，但容易被滥用。

函数模板和非模板函数之间的主要区别在于对实参隐式转换的支持度。非模板函数允许对实参进行广泛的隐式转换，从内建转换（例如整型提升）到用户自定义转换（借助未标以explicit的单参构造函数和转换操作符）。对于函数模板来说，由于编译器必须基于调用的实参类型来执行模板实参推导，因此只支持一些琐细的隐式转换，这包括涉及外层修饰（例如从T到const T，或从const T到T）、引用（例如从T到T&）以及从数组和函数退化的指针（例如从T[42]到T*）等情形。

这种区别所导致的实际效果是，函数模板需要比非模板函数更精确的匹配。这可以是好事，也可能是坏事，或者不过是让你大吃一惊。考虑下面的例子：

```
template <typename T>
void g( T a, T b ) { ... } // 这个g是一个模板
void g( char a, char b ) { ... } // 这个g是一个普通函数
//...
g( 12.3, 45.6 ); // 使用模板g
g( 12.3, 45 ); // 使用非模板的g!
```

第一个带有两个double实参的调用可以通过将double隐式地转换为char（合法但不可取）从而匹配非模板的g，然而，通过实例化模板g（此时T为double）可以得到一个更精确的匹配，因此选择的是模板版本的g。第二个调用带有一个double和一个int实参，将不会匹配模板g，因为编译器不会尝试对第二个实参进行从int到double的预定义转换（或者对第一个实参执行从double到int的转换），以便将T推导为double（或int）。因此，将调用非模板的g，使用从double到char以及从int到char的不吉利的预定义转换。

当面临各种不同的模板和非模板候选者时，挑选一个正确版本的函数（模板）是一个相当复杂的过程，许多值得信赖的编译器面对此情此景也会选择错误的函数（模板）或者

产生不当的错误。这同时也暗示了代码的维护者在理解希望调用哪一个版本的重载模板时，可能会遇到类似的困难。为了方便大家，当使用函数模板重载时，应该使事情对用户而言保持尽可能的简单。

对用户保持“简单”，意味着要对自己狠一些。在“模板实参推导[条款 57]”中，我们考虑了一个辅助函数，它机智地解决了一个费力且易犯错误的复杂类模板的特化问题：

```
template <typename A1, typename A2, typename R>
class PFun2 : public std::binary_function<A1,A2,R> {
    // 参见“模板实参推导[条款 57]”中的实现
};
```

不是强迫用户去直接特化这个怪物，我们提供了一个辅助函数，它用于执行模板实参推导和特化工作：

```
template <typename R, typename A1, typename A2>
inline PFun2<A1,A2,R> makePFun( R (*pf)(A1,A2) )
    { return PFun2<A1,A2,R>(pf); }
```

从语法上来说，这段代码相当复杂，但它简化了用户的工作，允许用户在面对声明为 `bool isGreater(int,int)` 的函数时，编写 `makePFun(isGreater)` 而不是 `PFun2<int,int,bool>(isGreater)`。

当然，我们也希望提供针对一元函数的程序：

```
template <typename A, typename R>
class PFun1 : public std::unary_function<A,R> {
public:
    explicit PFun1( R (*fp)(A) ) : fp_( fp ) {}
    R operator()( A a ) const
        { return fp_( a ); }
private:
    R (*fp_)(A);
};
```

以及一个辅助函数：

```
template <typename R, typename A>
inline PFun1<A,R> makePFun( R (*pf)(A) )
    { return PFun1<A,R>(pf); }
```

这是对函数模板重载的一个完美的应用。它简单，因为将会调用哪一个版本的 `makePFun` 不可能产生任何混淆（一个针对二元函数，另一个针对一元函数），同时由于我们为两个函数取了相同的名字，又使得这种程序学起来更容易，用起来更方便。

条款 59

SFINAE

当试图使用函数模板实参推导机制在多个重载函数模板和非模板函数中进行选择时，编译器可能会尝试一些失败的特化：

```
template <typename T> void f( T );
template <typename T> void f( T * );
//...
f( 1024 ); // 实例化第一个 f
```

尽管在第二个函数模板 `f` 中拿一个非零整数来替换 `T*` 是错误的，然而，只要发现了一个正确的替换，这个尝试过的替换就不会导致报错。在上例中，由于第一个 `f` 被成功地实例化了，因此不会报任何错误。这样，我们就引入了“替换失败并非错误（*substitution failure is not an error*）”的概念，Vandevoorde 和 Josuttis^① 给它起了个绰号：SFINAE。

SFINAE 是一个非常重要的属性，如果没有它，就很难去重载函数模板，因为实参推导和重载机制搅和在一起，将会导致很多对重载函数模板的使用是非法的。另外，SFINAE 的作用并非仅限于此，作为一种元编程技术，它也价值不菲。

回忆一下在“针对类型信息的特化[条款 52]”中开发的 `IsPtr` 程序。在那里，为了确定一个未知的类型是否为某种类型的指针，使用了模板局部特化技术。在这里，可以使用 SFINAE 来获得类似的结果：

```
typedef True char; // sizeof(True) == 1
typedef struct { char a[2]; } False; // sizeof(False) > 1
//...
template <typename T> True isPtr( T * );
False isPtr( ... );

#define is_ptr( e ) (sizeof(isPtr(e))==sizeof(True))
```

217

在这里，可以使用 `is_ptr` 来确定一个表达式的类型是否为一个指针，这是通过结合使用函数模板实参推导和 SFINAE 而达成的。如果表达式 `e` 为指针类型，编译器将会匹配函数

^① Vandevoorde 和 Josuttis 是 *C++ Templates* 一书的作者。这本书堪称模板大观。——译者注

模板 `isPtr`，否则它将会匹配带有省略号形参的非模板函数 `isPtr`。SFINAE 可以确保以非指针类型来匹配模板 `isPtr` 的尝试不会导致编译错误。

第二个有点奇妙的地方是在 `is_ptr` 宏中使用了 `sizeof`。可以注意到两个 `isPtr` 函数都没有提供定义。这是正确的，因为它们永远都没有被实际调用。`sizeof` 表达式中出现的函数调用，会导致编译器去执行实参推导和函数匹配工作，但它并不会真的去调用该函数。`sizeof` 操作符只对“可能被调用”的函数的返回类型的大小感兴趣。因而可以检查函数的返回类型的大小，以便确定该匹配哪一个函数。如果编译器选择了函数模板，那么表达式 `e` 的类型即为某种指针。

我们不必像在使用类模板局部特化实现 `IsPtr` 程序时所做的那样，分别处理 `const`、`volatile` 或 `const volatile` 指针等特殊情形。作为函数模板实参推导的一个组成部分，编译器将会忽略“一级”`cv` 修饰符（即 `const` 和 `volatile` 修饰符），也会忽略引用修饰符（参见“重载函数模板[条款 58]”）。类似地，我们也不必操心会发生这样的情况：将带有重载指针操作符的自定义类型误认为是一个指针类型。因为在函数模板实参推导过程中，编译器对实参只会施用一套极有限的转换规则清单，用户自定义转换并不在此清单之列。

可以注意到，这项技术与我们在“针对类型信息的特化[条款 52]”中使用模板局部特化来揭示类型信息具有一定的相似性。在那里，使用主模板作为一个“包罗万象”的装置，并使用完全特化或局部特化来侦测我们感兴趣的情形。在这里，使用一个带有省略号形参的函数来作为“包罗万象”的工具，并使用更精确的重载版本来捕获我们感兴趣的情形。实际上，类模板局部特化和函数模板重载在技术上有着极其密切的关系，C++ 标准其实就是根据它们中的一个来定义另一个的选择算法。

在熟悉了上面这个 `is_ptr` 例子后，从技术的角度而言，关于 SFINAE 实际上也就没有什么更多的东西可说了。然而，这项简单的技术可以以相当令人惊讶的方式进行使用，用于揭示关于类型和表达式在编译期的信息。让我们看一些（一点儿都不简单的）例子。

考虑这样一个问题：确定一个未知的类型是否为一个类？

```
template <typename T>
struct IsClass {
    template <class C> static True isClass( int C::* );
    template <typename C> static False isClass( ... );
    enum { r = sizeof(IsClass<T>::isClass<T>(0))
           == sizeof(True) };
};
```

撇开整洁不谈，这一次我们将 SFINAE 机制封装在一个类模板 `IsClass` 内，并且重载两个函数模板作为 `IsClass` 的静态成员。其中一个模板带有一个指向成员的指针的实参

(参见“指向类成员的指针并非指针[条款15]”)。由于一个字面0值可被转换为一个指向类成员的指针(对于函数模板来说也是如此),因此,如果T是一个类类型,那么第一个isClass将得到匹配。如果T不是一个类,那么SFINAE将会忽略第一次匹配尝试的错误,并选择带有省略号实参列表的isClass。就像is_ptr一样,我们可以通过检查函数返回类型的大小来查看哪一个函数被匹配了,从而确定T到底是不是一个类。

接下来的这个例子摘录自Vandevoorde和Josuttis的著作。假定希望知道某个特定的类类型中是否有一个名为“iterator”的嵌套类型(当然,这可以实现为查询任何嵌套的类型名字,未必非得是“iterator”):

```
template <class C>
True hasIterator( typename C::iterator const * );
template <typename T>
False hasIterator( ... );
#define has_iterator( C )\
    (sizeof(hasIterator<C>(0))==sizeof(True))
```

219

这个has_iterator程序从机制上说与IsClass是一致的,但这一次我们是访问一个未知类型的嵌套的类型名字(参见“采用typename消除歧义[条款49]”)。如果C具有这么一个嵌套的类型,我们就能够将字面0值转换为一个指向该类型的指针,否则将会匹配那个“包罗万象”的版本。

最后,让我们看一个由Andrei Alexandrescu^①提供的技巧:给定两个未知类型T1和T2,能否将T1转换为T2?注意,这个机制不仅能够侦测预定义转换,也能够侦测用户自定义的转换:

```
template <typename T1, typename T2>
struct CanConvert {
    static True canConvert( T2 );
    static False canConvert( ... );
    static T1 makeT1();
    enum { r = sizeof(canConvert( makeT1() )) == sizeof(True) };
};
```

正如在“针对类型信息的特化[条款52]”中看到的Heap实现一样,能够基于可在编译期静态确定的信息而提供专用目的的实现,通常具有很大的灵活性或效率优势。通过使用SFINAE以及其他元编程技术,我们就能够询问诸如此类的问题:“这个未知的类型是一个指向类类型的指针吗?——该类类型具有一个嵌套的“iterator”类型名字,后者又可被转换为std::string。”

220

① Andrei Alexandrescu 是 *Modern C++ Design* 一书的作者。本例摘自该书第2章“Techniques”并稍作了修改。

条款 60

泛型算法

一个泛型算法就是一个以这种方式设计的函数模板：根据使用的上下文，它可以在编译期容易且有效地进行定制。让我们看一个函数模板，它不满足这个苛刻的标准，因此不是一个适当的泛型算法：

```
template <typename T>
void slowSort( T a[], int len ) {
    for( int i = 0; i < len; ++i ) // 对于每一对元素……
        for( int j = i; j < len; ++j )
            if( a[j] < a[i] ) { // 如果次序颠倒的话
                T tmp( a[j] ); // 则执行交换
                a[j] = a[i];
                a[i] = tmp;
            }
}
```

这个模板可以用于对一个对象数组进行排序，只要这种对象支持“<”比较操作符和复制操作即可。例如，我们可以对来自“赋值和初始化并不相同[条款 12]”中的 String 对象数组进行排序：

```
String names[] = { "my", "dog", "has", "fleece" };
const int namesLen = sizeof(names)/sizeof(names[0]);
slowSort( names, namesLen ); // 排啊排啊……最后终于排好了！
```

关于slowSort的第一个抱怨可能是它太慢了。这个观察是正确的，但暂请原谅slowSort的 $O(n^2)$ 时间复杂度吧，让我们将注意力集中在它的泛型设计方面。

221

第一个观察是slowSort中的交换实现对于String类型（以及其他许多类型）来说不够理想。String类具有自己的成员swap，与通过复制一个临时String对象而完成的交换相比，它更快，并且更加异常安全。一个更好的实现方式就是直接表达我们的意思：

```
template <typename T>
```

```

void slowSort( T a[], int len ) {
    for( int i = 0; i < len; ++i ) // 对于每一对元素……
        for( int j = i; j < len; ++j )
            if( a[j] < a[i] ) // 如果次序颠倒的话
                swap( a[j], a[i] ); // 则执行交换
}

```

我们仍然不能调用String的swap成员函数，但是，如果String类的作者还提供了一个非成员版本的swap，那么该版本的swap将会被调用：

```

inline void swap( String &a, String &b )
{ a.swap( b ); }

```

设想如果不存在这样一个非成员版本的swap，情况会如何？如果是那样的话，处境不会更糟糕，因为无论如何我们还可以调用标准库中的swap，它正好做了和我们在最初版本的slowSort中手工编码一样的事情。实际上，在这种情况下，仍然比最初的情况好得多，因为这个新版的slowSort实现更短小，更简单，更易理解。更重要的是，如果某人最终为String实现了一个高效的非成员的swap，我们就可以自动地获得该项改善，这样的代码维护工作显然是可以忍受的。

现在来考虑使用<操作符来比较数组的元素。这可能是希望对数组元素进行排序的最常见的方式了（即从最小到最大排序），但我们可能也希望按降序或某种特殊的顺序来排序。此外还存在这样的可能：有些我们希望对其排序的数组中的对象，要么不支持<操作符，要么具有好几个截然不同的类似“小于”操作符的候选函数。在“STL 函数对象[条款 20]”中已经看到了这样的—一个类型，即State类：

```

class State {
public:
    //...
    int population() const;
    float aveTempF() const;
    //...
};

```

223

在“STL 函数对象[条款 20]”中采用的方式是：实现“可用来替换<操作符”的函数和函数对象，但只有当泛型算法被设计成可以接受这样的—一个参数时，这种方式才可行：

```

template <typename T, typename Comp>
void slowSort( T a[], int len, Comp less ) {

```

```

    for( int i = 0; i < len; ++i ) // 对于每一对元素……
        for( int j = i; j < len; ++j )
            if( less( a[j], a[i] ) ) // 如果次序颠倒
                swap( a[j], a[i] ); // 则执行交换
    }
    //...
    State union{50};
    //...
    slowSort( union, 50, PopComp() );

```

如果要经常采用 < 操作符来执行 slowSort 操作，那么对 slowSort 进行重载是一个好主意，这样，既可以采用 < 操作符来调用它，也可以使用一个特殊用途的比较操作符来调用它。

最后要指出的是，遵从约定和惯例总是一个好主意，对于泛型算法来说尤其如此。我们还有理由批评 slowSort，因为它限制所排序的实参必须是数组，要知道还有很多其他种类的容器或数据结构可能需要排序。碰到这样的情况，如果有任何疑问，直接参照标准库的做法：

```

template <typename For, typename Comp>
void slowSort( For b, For e, Comp less ) {
    for( For i( b ); i != e; ++i ) // 对于每一对元素……
        for( For j( i ); j != e; ++j )
            if( less( a[j], a[i] ) ) // 如果次序颠倒
                swap( a[j], a[i] ); // 则执行交换
    }
template <typename For>
void slowSort( For b, For e ) {
    for( For i( b ); i != e; ++i ) // 对于每一对元素……
        for( For j( i ); j != e; ++j )
            if( a[j] < a[i] ) // 如果次序颠倒
                swap( a[j], a[i] ); // 则执行交换
    }
    //...
    std::list<State> union;
    //...
    slowSort( union.begin(), union.end(), PopComp() );
    slowSort( names, names+namesLen );

```

其中，我们将笨拙的数组接口替换为一个更标准且更灵活的、与 STL 相容的迭代器接

口。现在可以心安理得地称 `slowSort` 是一个泛型算法了，而不仅仅称其为一个函数模板。

从这个例子中可以获得一个非常重要的经验，那就是复杂的软件设计几乎总是群体努力的结果。同样的道理，代码也应该以这种方式进行设计，即尽可能利用同事的专家经验，同时还应该尽量保持不受代码维护的影响，即便那些代码维护工作不在你的控制之下。经过改善的 `slowSort` 算法是此类正确设计的一个好例子。它在一个尽可能高的概念层来执行单纯的、易于理解的操作。更精确地说，`slowSort` 集中精力处理排序算法，并将交换和比较操作“转包”给其他能够做得更好的人。这种方式允许你（假设是一名排序专家），利用任何人针对被排序的元素类型设计的专家方案，来增强你的排序专家技能。你们二位也许素未谋面，但通过适当的设计就可以紧密协作，就好像你们俩共用一台工作站似的。此外，如果有朝一日出现了改善的 `swap` 函数功能，`slowSort` 将会自动启用改善的版本，可能连你自己都不知道。这种“不知情”很有好处，俗话说得好，“无知者无畏”（这类似于恰当的多态设计的情形。参见“**Command 模式与好莱坞法则**[条款 19]”）。

条款 61

只实例化要用的东西

在 C 和 C++ 中，如果没有调用一个已声明的函数（或试图获取其地址），那么就不需要定义它。对于类模板的成员函数来说存在类似的情形。如果实际上并没有调用一个模板的成员函数，那么该成员就不会被实例化。

对于减少代码的规模来说，这显然是一个方便的属性。如果一个类模板定义了大量的成员函数，但是只使用了其中的两、三个，就无需为所有未使用的函数付出代码空间的代价。

从这个规则可以得出的一个更重要的结论：用来特化一个类模板的模板实参，并不一定要使得该类模板的所有成员函数都能被合法地实例化。有了这个规则，我们就能够编写灵活的类模板，它可以处理范围广泛的实参，即使一些实参可能导致对某些成员函数进行错误的实例化。如果没有真正调用这些错误的函数，它们就不会被实例化，也就不会导致出现错误。这与 C++ 语言中的许多领域都是一致的：在它们真正成为问题之前，潜在的问题并不会被标记为错误。也就是说，在 C++ 中，你可以“想入非非”，只要不付诸行动即可！

考虑一个简单的、固定大小的数组模板：

```
template <typename T, int n>
class Array {
public:
    Array() : a_( new T[n] ) {}
    ~Array() { delete [] a_; }
    Array( const Array & );
    Array &operator =( const Array & );
    void swap( Array &that ) { std::swap( a_, that.a_ ); }
    T &operator []( int i ) { return a_[i]; }
    const T &operator []( int i ) const { return a_[i]; }
    bool operator ==( const Array &rhs ) const;
    bool operator !=( const Array &rhs ) const
```

```

        { return !(*this==rhs); }
private:
    T *a_;
};

```

这个容器的行为酷似预定义数组，它具有常用的索引操作，同时还提供了一些预定义数组所没有提供的更高级的操作，比如交换和相等性比较（因为篇幅的缘故，省略了一些关系操作符）。让我们看一个 `operator ==` 的实现：

```

template <typename T, int n>
bool Array<T,n>::operator ==( const Array &that ) const {
    for( int i = 0; i < n; ++i )
        if( !(a_[i] == that.a_[i]) )
            return false;
    return true;
}

```

我们知道两个被比较的数组具有同样数目的元素，因为它们具有同样的类型并且数组的大小亦作为模板参数之一提供，因此只要对元素成对执行比较即可。如果发现有任何一对元素不相等，那么两个 `Array` 对象就是不相等的。

```

Array<int,12> a, b;
//...
if( a == b ) // 调用a.operator ==(b)
//...

```

对 `Array<int,12>` 对象使用 `==` 操作符时，编译器实例化 `Array<int,12>::operator==`，这可以正确地编译。如果没有对类型为 `Array<int,12>` 的对象使用 `==`（或 `!=` 操作符，它调用了 `==` 操作符），就不会实例化那个成员函数。

226

使用一个未定义 `==` 操作符的类型来实例化 `Array` 时，将会发生一个有意思的情形。举个例子，假定 `Circle` 类型没有定义（或从基类那里继承）一个 `operator ==`：

```

Array<Circle,6> c, d; // 没问题！
//...
c[3].draw(); // OK

```

至此为止，一切都好。我们尚未直接或间接地对 `Array<Circle,6>` 对象使用 `==` 操作，因此函数 `operator ==` 就不会被实例化，也就不会发生错误。

```

if( c == d ) // 错误！

```

现在我们就碰到问题了。编译器尝试去实例化 `Array<Circle,6>::operator ==`，从而函数实现就会试图拿不存在的 `==` 操作符来比较两个 `Circle` 对象，因而发生编译期错误。

这项技术通常用于类模板的设计中，从而使类模板尽可能灵活，但又不会灵活得过了头^①。

注意这种田园牧歌式的情形不会发生于显式实例化一个类模板时：

```
template Array<Circle,7>; // 错误!
```

这个显式实例化指示符告诉编译器使用实参 `Circle` 和 `7` 去实例化 `Array` 及其所有成员，从而导致当实例化 `Array<Circle,7>::operator ==` 时发生编译期错误。呃，真是自讨苦吃。

227

^① “不会灵活得过了头”，是因为错误发生于编译期而非运行期。——译者注

条款 62

包含哨位

在产品级的 C++ 应用中往往会用到大量的头文件，并且许多头文件还包含其他头文件。在这些情况下，同一个头文件被多次间接地包含于同一个编译单元中是很常见的，甚至在一个大型复杂的应用中，同一个头文件被包含于同一个编译单元中好几百次也并非不常见。考虑一个简单的情形，头文件 `hdr2.h` 包含另一个头文件 `hdr1.h`，并且头文件 `hdr3.h` 也包含头文件 `hdr1.h`。这样，如果头文件 `hdr2.h` 和头文件 `hdr3.h` 被包含于同一个源文件中，那么 `hdr1.h` 就会被包含两次。一般而言，这种多次包含的结果并非我们的本意，最终会导致多次定义错误。

鉴于此，C++ 头文件几乎普遍采用预处理技术来防止头文件内容在一个编译单元中出现不止一次，而不管该头文件实际上被 `#include` 了多少次。考虑头文件 `hdr1.h` 的内容：

```
#ifndef HDR1_H
#define HDR1_H
// 头文件的实际内容……
#endif
```

当头文件 `hdr1.h` 第一次被 `#include` 进一个编译单元时，预处理符号 `HDR1_H` 尚未定义，因此 `#ifndef`（即“if not defined（如果尚未定义）”）预处理条件允许预处理 `#define` 指令以及其余的头文件内容。当下一次 `hdr1.h` 出现于同一个编译单元时，由于符号 `HDR1_H` 已经定义了，因此 `#ifndef` 语句就会阻止对该头文件内容的重复包含。

只有当一个头文件预处理符号（本例中为 `HDR1_H`）的确和一个对应的头文件（本例中为 `hdr1.h`）相关联时，这项技术才能工作。因此，建立一个标准的、简单的命名约定，从而允许根据被守卫的头文件名字，来构建用于包含哨位（include guard）中的预处理符号的名字，是非常重要的。

229

除了可以防止发生错误外，使用包含文件哨位还有助于加快编译速度，这是通过允许编译器略过任何已经被翻译过的头文件的内容而做到这一点的。不幸的是，在一些复杂的场合下，一个给定的编译单元中多个头文件可能会出现许多次，从而导致“打开一个头文件、验证 `#ifndef` 并扫描结束符 `#endif`”这个过程非常耗时。在某些情况下，采用冗余

的包含哨位可以相当显著地加快编译速度：

```
#ifndef HDR1_H
#include "hdr1.h"
#endif
```

其中，我们不是简单地 `#include` 头文件，而是通过测试头文件里相同的哨位符号来守卫这个 `#include`。这当然是冗余的，因为当头文件第一次被包含时，同一个条件（在这个例子中为 `#ifndef HDR1_H`）将被测试两次，即发生于此处的 `#include` 之前，以及该头文件自身之内。然而，对于后续的包含来说，这个冗余的哨位可以防止 `#include` 指令被执行，从而防止头文件被不必要地打开和扫描。对使用冗余的包含哨位并不像使用简单的包含哨位那样常见，但在某些情况下，尤其对于大型应用程序来说，使用它们可以成倍地加快编译速度。

条款 63

可选的关键字

尽管关于到底用不用这些关键字还有其他方面的争论，但从 C++ 语言的视角来看，有一些关键字的使用确实是可选的。

最常见的混淆出现于在重写了基类虚成员函数的派生类成员函数中可选地使用 `virtual` 关键字：

```
class Shape {
public:
    virtual void draw() const = 0;
    virtual void rotate( double degrees ) = 0;
    virtual void invert() = 0;
    //...
};

class Blob : public Shape {
public:
    virtual void draw() const;
    void rotate( double );
    void invert() = 0;
    //...
};
```

成员函数 `Blob::draw` 重写了基类虚函数 `draw`，因此它也是虚拟的。其中对关键字 `virtual` 的使用完全是可选的，使用与否对程序的含义没有任何影响。一个常见的误解是，遗漏使用 `virtual` 关键字将会阻止在下级派生类中进一步重写该函数，事实并非如此：

```
class SharpBlob : public Blob {
public:
    void rotate( double ); // 重写 Blob::rotate
    //...
};
```

注意，当重写 `Blob::invert` 这样的纯虚函数时，`virtual` 关键字同样是可选的。总

之，在一个重写的派生类函数中，virtual 关键字出现与否完全是可选的，对于程序的含义没有丝毫影响。

在一个重写的派生类函数中，省略 virtual 关键字是否为一个良好的编程实践，意见分为两派。一些权威人士声称使用本质上并不必要的 virtual 关键字，有助于为人们提供有关派生类函数性质的文档。另外一些专家则声称这种做法纯粹是浪费精力，并可能导致一个非重写的派生类函数不留神被误标为 virtual。不管赞成哪一种意见，最好保持一致，即要么对每一个重写的派生类函数使用 virtual 关键字，要么全部省略不写。

当声明 operator new、operator delete、array new 以及 array delete 成员时（参见“特定于类的内存管理[条款 36]”），static 关键字是可选的，因为这些函数是隐式静态的：

```
class Handle {
public:
    void *operator new( size_t ); // 隐式 static
    static void operator delete( void * );
    static void *operator new[]( size_t );
    void operator delete[]( void * ); // 隐式 static
};
```

一些权威人士声称，在这一点上应该尽可能明确，即总是将这些函数显式地声明为 static。另外一些人士则认为，如果使用或维护此类 C++ 代码的人不知道这些函数是隐式静态的，那么他们根本不配使用或维护这些代码。在这里使用 static 纯粹是浪费精力，程序没有义务为语言的语义“夹带一些纸条”。我的看法比较中庸，如前所述对 virtual 的可选使用一样，不管对可选的 static 关键字的使用抱着什么样的看法，最重要的是要保持一致。也就是说，这四个函数要么都显式地声明为 static，要么全不使用 static 关键字。

在一个模板的头部，关键字 typename 和 class 可以互换使用，表示一个模板参数是一个类型名字，在这种上下文中，二者含义无任何区别。然而，许多专家级的 C++ 程序员使用 typename 向人们指明该模板实参可以是任何类型，而使用 class 关键字来指明该类型实参必须是一个类类型：

```
template <typename In, typename Out>
Out copy( In begin, In end, Out result );

template <class Container>
```

在“古”时候，`register`关键字用于向编译器说明某个变量将会频繁地使用（这是从程序员的角度来看），因此建议将其放置于一个寄存器里。而且，试图获取一个声明为`register`存储类的变量的地址是非法的。然而，尽管早期的编译器作者意识到他们的编程同行们对于什么变量应该存储在寄存器中完全凭主观臆断，但是对`register`的建议有时的确会予以考虑，而现今的编译器则整齐划一地忽略了程序员的这个建议。所以说，在C++中，使用`register`与否对于程序的含义没有任何影响，对于程序的效率（一般）也没有影响。

`auto`关键字可以用于指示一个自动变量（一个函数实参或一个局部变量）是自动的。不必为之烦神。

不过我坦诚这一点：`register`和`auto`可以用在一些模糊难解的场合下，借以消除一些编写得极其糟糕的代码之语法歧义。不过话又说回来，在这些情况下，正确的做法是编写更好的代码并且避免使用这些关键字。

参考文献

- Alexandrescu, Andrei. *Modern C++ Design*. Addison-Wesley, 2001.
- Dewhurst, Stephen C. *C++ Gotchas*. Addison-Wesley, 2003.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- Josuttis, Nicolai M. *The C++ Standard Library*. Addison-Wesley, 1999.
- Meyers, Scott. *Effective C++*, Third Edition. Addison-Wesley, 2005.
- Meyers, Scott. *Effective STL*. Addison-Wesley, 2001.
- Meyers, Scott. *More Effective C++*. Addison-Wesley, 1996.
- Sutter, Herb. *Exceptional C++*. Addison-Wesley, 2000.
- Sutter, Herb. *More Exceptional C++*. Addison-Wesley, 2002.
- Sutter, Herb. *Exceptional C++ Style*. Addison-Wesley, 2005.
- Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2005.
- Vandevoorde, David, and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley, 2003.
- Wilson, Matthew. *Imperfect C++*. Addison-Wesley, 2005.

索引

索引中的页码为英语原书页码,与书中页边标注的页码一致。

另请参考第195页“代码示例索引”。

->*(dash angle bracket asterisk), pointer-to-member operator
(指向成员的操作符),58

--> operator, overloading(操作符,重载),145-146,147-148
() (parentheses)(括号)

function call operator(函数调用操作符),58

grouping in declarations,61

grouping with pointer-to-member operators,58

*(asterisk) operator, overloading(操作符,重载),145-146,147-148

. * (dot asterisk), pointer-to-member operator
(指向成员的操作符),58

1984 (veiled reference), 224

A

ABC (abstract base class)(抽象基类),113

abort,117,140

Abstract data type(抽象数据类型). 见 data abstraction

Access protection(访问保护),88, 111, 113-115

Accessor functions(访问函数). 见 get/set interfaces

Address arithmetic(地址算术). 见 pointer arithmetic

ADL (argument dependent lookup) (实参相依的查找),89-90

Aliases(别名). 参见 references

Allocating arrays(数组分配),127-129

Allocators, rebind convention(分配器, rebind 约定),180

Anonymous namespaces(匿名名字空间),84-85

Argument dependent lookup (ADL) (实参相依的查找),89-90

Arguments, templates(模板实参),153-154

Array declarators, pointers to(指向数组声明符的指针),61-62

Array formal arguments(数组形参),17-19

Array index calculation(数组下标计算). 见 pointer arithmetic

Arrays(数组)

allocation/deallocation(分配/归还),127-129

auto_ptr references(auto_ptr引用),148

of class objects(类对象的数组),120-121

decay(退化),17

as function arguments(作为函数实参). 见 array formal arguments

memory management(内存管理),127-129

of pointers(指针的数组),25

references to(数组引用),62

sorting(排序),221-222

Assignment(赋值). 见 copying, initialization

computational constructor(计算性的构造函数),43

construction(构造),42

copying(复制),45-47

destruction(析构),42-43

exception safe copy(异常安全复制),46

versus initialization(vs. 初始化),41-43

user-defined types(用户自定义的类型),41-43

virtual copy(虚复制),47

and virtual function table pointers(与虚函数表指针),38

Asterisk (*) operator, overloading (* 操作符, 重载),145-146
147-148

Audit trails, resource control(审计索引,资源控制). 见 RAII

auto keyword(auto关键字),233

auto_ptr

array references(数组引用),148

as container elements(作为容器元素),148

conversions(转换),147-148

description(描述),147-148

operator overloading(操作符重载),147

versus smart pointers(vs. 智能指针),148

B

Base classes(基类)

- abstract bases, manufacturing(制造抽象基类),113-115
- forcing to abstract base(抽象基类),113-115
- implementation guidelines(实现方针),77-79
- member function, determining(成员函数,确定),77-79
- polymorphic(多态),3-5
- Template Method,77-79

Body objects(本体对象),117

Boedigheimer, Kim, xvii

Bradbury, Ray (veiled reference),50

The Brother,xv,81,147

C

Callbacks(回调). 另见 Command 模式

- definition(定义),67
- "don't call us, we'll call you"(不要call我们,我们会call你),68
- framework code(框架代码),68
- function objects as(函数对象作为回调),67-70,另见 Command 模式
- function pointers(函数指针),50-51
- Hollywood Principle(好莱坞法则),68

Capability queries(能力查询),93-95

Cast operators(转型操作符)

- casting down(向下转型)
 - an inheritance hierarchy(继承层次结构),30-31
 - from a pointer to a base class(指向基类的指针),31
 - to a reference type(引用类型),32

const qualifiers, adding/removing(const 修饰符,添加/移除),29-30

const_cast, 29-30

conversion across a hierarchy(层次结构中的转换),94-95

cross-cast(横向转型),94-95

dynamic_cast, 31-32

function style(函数风格),29

new style(新风格)

description(描述),29-32

versus old-style(vs.旧风格),29

old-style versus new style(旧风格 vs.新风格),29

reinterpret_cast, 31

static_cast, 30-32

type qualifiers, changing(类型修饰符,改变),29-30,30-31

volatile qualifiers, adding/removing(volatile 修饰符,添加/移除),29-30

Class layout(类的布局)

- assignment, and virtual function table pointers(赋值,虚函数表指针),38
- contravariance(逆变性),54-55
- covariant return types(协变返回类型),109
- member offsets(成员偏移量),39
- pointer comparison(指针比较),97-98
- virtual functions(虚函数),37
- virtual inheritance(虚拟继承),37-38
- what you see is what you get(所见即所得),37

Class members, pointers to versus pointers(类的成员,指向类成员的指针和指针),53-56

Class objects(类对象). 见 objects

Classes(类)

- handle, exception safe swaps(句柄,异常安全的交换),46
- interface(接口),93-94

Cloning objects(克隆对象),99-101. 另见 assignment; copying; initialization; Prototype

Command pattern(Command 模式). 另见 callbacks

Communication, with other programmers(与其他程序员交流)

- data abstraction(数据抽象),2
- design patterns(设计模式),7-8
- identifiers in template declarations(模板声明中的标识符),201-202
- versus ignorance(vs.忽略),224
- overloading(重载),214-215
- typedefs(typedef),62

Comparator operations(比较操作),71

Comparators, STL function objects as(STL 函数对象作为比较器),72-73

Complete specialization(完全特化). 见 explicit specialization

Computational constructor(计算性的构造函数),41,43

Const member functions(常量成员函数)

- lazy evaluation(缓式求值),34
- logically const(逻辑常量),35
- meaning of(常量成员函数的含义),33-36
- modifying objects(修改对象),34
- overloaded index operator(重载索引操作符),35-36

Const pointers, *versus* pointers to const(常量指针与指向常量的指针), 21-23

const qualifiers, adding/removing(const 修饰符, 添加/移除), 29-30

const_cast operator(const_cast 操作符), 29-30

Construction(构造)

assignment(赋值), 42

copying(复制), 45-47

Construction order(构造顺序), 141-142

Constructors(构造函数)

calling(调用), 119-121

description(描述), 143-144

exception safety(异常安全), 143-144

operator overloading(操作符重载), 143-144

placement new(定位 new), 119-121

protected(受保护的), 114-115

virtual(虚构造函数), 99-101

Container elements, auto_ptr as(auto_ptr 作为容器元素), 148

Contract, base class as(基类作为承包者), 4-5

Contravariance(逆变性)

class layout(类布局), 54-55

member templates(成员模板), 174

pointers to data members(指向数据成员的指针), 54-55

pointers to member functions(指向成员函数的指针), 54-55, 58

Conventions(约定), 另见 idioms

anonymous temporary function object
(匿名临时函数对象), 73

class vs. typename(class vs. typename), 232-233

copy operations(复制操作), 45

exception safety axioms(异常安全公理), 131-133

generic programming(泛型编程), 191, 193, 207, 223

in generic programming(泛型编程), 170

multilevel pointers(多级指针), 26

naming conventions(命名约定), 88, 229

placement of const qualifier(const 修饰符的位置), 21-22

rebind convention for allocators(分配器的 rebind 约定), 180
and the STL(和 STL), 12

STL (standard template library)(标准模板库), 11

traits and promulgation(), 193, 197

unnecessary static(不必要的 static), 232

unnecessary virtual(不必要的 virtual), 232

Conversions, auto_ptr(转换, auto_ptr), 147-148

Copying(复制), 另见 assignment; cloning; initialization

address of non-const to pointer to const(将一个 non-const 对象的地址复制给一个指向 const 的指针), 22

assignment(赋值), 45-47

class objects(类对象), 38

construction(构造), 45-47

objects(对象), 38

Covariance(协变性), 174

Cranberries(蔓越桔), 183

Cross-cast(横向转型), 94-95

D

Dash angle bracket asterisk (->*), pointer-to-member
operator(指向成员的指针操作符), 58

Data abstraction(数据抽象), 1-2

Data hiding(数据隐藏), 见 access protection; data abstraction

Deallocating arrays(数组归还), 127-129

Decay(退化)

arrays(数组), 17, 25

functions(函数), 17, 72

Declaring function pointers(声明函数指针), 49

Delta adjustment of class pointer(类指针的 delta 调整),
58, 98, 108

Design patterns(设计模式)

description(描述), 7-10

essential parts(关键部分), 8-10

Factory Method, 103-106, 108-109

microarchitectures(微架构), 10

Template Method

versus C++ templates(Template Method 与 C++ 模板), 77

description(描述), 77-79

wrappers(包装器), 8

Destruction(析构)

assignment(赋值), 42-43

order(顺序), 141-142

RAII, 140

restricting heap allocation(禁止或强制使用堆分配), 117

Disambiguation(消除歧义)

auto keyword(auto 关键字), 233

register keyword(register 关键字), 233

with template(采用 template 消除歧义), 179-181

templates(模板),179-181
 with typename(采用 typename 消除歧义),169-172
 “Don’t call us, we’ll call you,”(不要 call 我们,我们会 call 你),68,79
 Dot asterisk (.), pointer-to-member operator(指向成员的指针操作符),58
 Down casting(向下转型).另见 cast operators
 runtime type information(运行期类型信息),31
 safe,31
 dynamic_cast operator(dynamic_cast 操作符),31-32

E

e,65,66
 Element type, determining(确定元素类型),189-191
 Embedded type information(嵌入的类型信息),189-191
 Exception safety(异常安全)
 axioms(公理),131-133
 catching(捕获),135-137
 constructors(构造函数),143-144
 copy assignment(复制赋值),46
 exceptions(异常),143-144
 functions(函数),135-137
 new operator(新式操作符),143-144
 safe destruction(安全析构),132
 swap throwing(交换抛出异常),133
 synchronous exceptions(同步异常),131-132
 throwing(抛出),135-137
 Exceptions, memory allocation(异常,内存管理),143-144
 exit and destruction(exit 与析构),117,140
 Explicit specialization(显式特化),155-159, 183-187
 Expressions(表达式)
 const qualifiers, adding/removing(添加/移除 const 修饰符),29-30
 volatile qualifiers, adding/removing(添加/移除 volatile 修饰符),29-30

F

Fahrenheit 451 (veiled reference),50
 File handles, resource control(文件句柄,资源控制).见 RAII
 Fire safety advice(救火安全建议),50
 Framework code, callbacks(回调),68
 Function declarators, pointers to(指向函数声明符的指针),61-62

Function objects(函数对象).另见 function pointers;
 smart pointers
 as callbacks(回调),67-70.另见 Command pattern
 description(描述),63-66
 integrating with member functions(与成员函数集成),66
 Function overloading(函数重载).另见 operator overloading
 function templates(函数模板),213
 generic algorithms(泛型算法),223
 infix calls(中缀调用),90
 overloaded index operator(重载索引操作符),35-36
 pointers to overloaded functions(指向重载函数的指针),51
 scope(作用域),88
 SFINAE,217
 taking address of(取地址),51

Function pointers(函数指针).另见 function objects
 callbacks(回调),50-51
 declaring(声明),49
 description(描述),49-51
 generic(通用),49-50
 to inline functions(指向内联函数),50
 to non-static member functions(指向非静态成员函数),50
 to overloaded functions(指向重载函数),51
 virtual(虚函数指针),64-66

Function style(函数风格),29

Function templates(函数模板)
 generic algorithm(泛型算法),221-224
 versus nontemplate functions(与非模板函数),213
 overloading(重载),213-215
 template argument deduction(模板实参推导),209-212

Functionoid(函数标识符).见 function objects

Functions(函数)

 arguments from arrays(数组参数).见 array formal arguments
 decay(退化),17,72
 multilevel pointers(多级指针),25,27
 references to(引用),62
 selecting right version(选择正确的版本),214
 static linkages, avoiding(避免静态连接),85

Functor(仿函数).见 function objects

G

Generic function pointers(通用函数指针),49-50
 Gentleman Cambrioleur,196

get/set interfaces(获取/设置操作),1

Global scope, namespaces(全局作用域,名字空间),81-85

Graphical shapes, resource control(绘图形状,资源控制).见 RAII

Guarding against multiple inclusions(以多重包含设置哨位),
229-230

H

Handle/body idiom(句柄/本体惯用法),117

Handle classes(句柄类)

exception safe swaps(异常安全的交换),46

RAII,139

restricting heap allocation(禁止或强制使用堆分配),117

Header files, multiple inclusion(头文件,多重包含),229-230

Heap(堆)

algorithms(算法),155

allocation, restricting(禁止或强制使用堆分配),117-118

class template explicit specialization(类模板显式特化),
155-159

Heinlein, Robert (veiled reference),76

Hitchhiker's Guide to the Galaxy(veiled reference),211,213

Hollywood principle(好莱坞法则),68,79,224

I

Idioms(惯用法/术语)

assumption of non-throwing deletion(有关不抛出异常的
delete假设),136

checking for assignment to self(检查是否自身赋值),47

computational constructor(计算性的构造函数),43

to create abstract base class(创建抽象基类),113-115

exception safety axioms(异常安全公理),131-133

function object(函数对象),63

getting current new_handler(获得当前的new_handler),51

intent of dynamic_cast to reference(对引用进行
dynamic_cast),32

meaning of assignment(赋值的含义),46

ordering code for exception safety(异常安全的代码),136

partitioning code to avoid RTTI(划分代码以避免RTTI),94

to prohibit copying(禁止复制),111

RAII,139-142

reference to pointer formal argument(指针形参引用),26

resource acquisition is initialization(资源获取即初始化),

139-142

to restrict heap allocation(禁止或强制使用堆分配),117-118

result of assignment vs. initialization(赋值与初始化的
结果),46

robust computation of array size(数组大小的强健
计算方式),17

smart pointer(智能指针),145-147

STL function object(STL 函数对象),72-73

violation of copy operation idioms(违背复制操作
惯用法),148

virtual constructor(虚构造函数),100

Ids, templates(模板标识符),154

Ignorance(忽略)

callbacks(回调),67

healthful aspects(好的方面),5,12,224

Java programmers(Java 程序员),37

of object type(对象类型的忽略),100-101

and templates(与模板),180

Initialization(初始化).另见 assignment; copying

argument passing(参数传递),41

versus assignment(和赋值),41-43

catching exceptions(捕获异常),41

declaration(声明),41

function return(函数返回),41

Initialization order(初始化顺序).见 construction order

Inline functions, pointers to(指向内联函数的指针),50

Instantiation(实例化)

template member functions(模板成员函数),225-227

templates(模板),154

Integers as pointers(整数作为指针)

new cast operators(新式转型操作符),29,31

placement new(定位 new),119

pointer arithmetic(指针算术),151

Interface class(接口类),65,93

J

Java

function and array declarators(函数和数组声明符),62

good natured poke at,37

interface classes(接口类),93

member function lookup(成员函数查找),88

Josuttis, Nicolai, 217-220

K

Keywords, optional(可选的关键字), 231-233

Koenig lookup(Koenig 查找). 见 ADL

(argument dependent lookup)

Kyu Sakamoto, reference to(引用/参考 Kyu Sakamoto),
42, 68, 70

L

Lazy evaluation(缓式求值), 34

Lippman, Stan, 141

Logically const(逻辑上为 const), 35

Login sessions, resource control(登录会话, 资源控制). 见 RAII
Lupin, Arsene, 196

M

Managers, gratuitous swipe at(无端地讽刺经理), 10

Martin, Steve (veiled reference), 46

McFerrin, Bobby (veiled reference), 179

Member functions(成员函数)

function matching errors(函数匹配错误), 87-88

integrating function objects(整合函数对象), 66

lookup(查找), 87-88

pointers to(……的指针)

contravariance(逆变性), 54-55

declaration syntax(声明语法), 57-58

integrating with function objects(与函数对象整合), 66

operator precedence(操作符优先级), 58, 61

versus pointers(与指针), 57-59

simple pointer to function(指向函数的简单指针), 58

virtualness(虚拟性), 58

roles(作用), 77-78

templates(模板), 173-177

Member object(成员对象). 见 class layout

Member offset(成员偏移量). 见 class layout

Member operator functions, overloading non-member operators
(成员操作符函数, 重载非成员操作符), (91-92)

Member specialization(成员特化), 165

Member templates(成员模板), 173-177

Memory(内存)

class-specific management(特定于类的内存管理), 123-126

heap allocation, restricting(禁止或强制使用堆分配),
117-118

resource control(资源控制). 见 RAII

Memory management, arrays(内存管理, 数组), 127-129

Multidimensional arrays(多维数组)

array formal arguments(数组形参), 18-19

function and array declarators(函数和数组声明符), 61

pointer arithmetic(指针算术), 150

N

Names, templates(模板名字), 154

Namespaces(名字空间)

aliases(别名), 84

anonymous(匿名), 84-85

continual explicit qualification(连续的显式限定), 82-83

description(描述), 81-85

names(名字)

declaring(声明), 82

importing(导入), 83

using declarations(using 声明), 84

using directives(using 指令), 82-83

Nested names, templates(嵌套名字, 模板), 179

Network connections, resource control

(网络连接, 资源控制). 见 RAII

New cast operations(新式转型操作符).

见 cast operators, new style

new operator(new 操作符)

description(描述), 143-144

exception safety(异常安全), 143-144

versus operator new,(与 operator new), 119, 123, 143

operator overloading(操作符重载), 143-144

New style cast operators, versus old-style

(新式与旧式转型操作符), 29

Non-static member functions, pointers to

(指向非静态成员函数的指针), 50

Nontemplate functions versus function templates

(非模板函数与函数模板), 213

O

Objects(对象)

alternate names for(对象的替代名字). 见 aliases; references

- arrays of(数组), 120-121
 - capability queries(能力查询), 93-95
 - changing logical state(改变逻辑状态), 33-36
 - cloning(克隆), 99-101
 - copying(复制), 38
 - copying, prohibiting(禁止复制), 111
 - creating, based on existing objects
 - (基于已经存在的对象进行创建), 103-106
 - heap allocation, restricting(禁止或强制使用堆分配), 117-118
 - integrating member functions with function objects
 - (将成员函数与函数对象整合), 66
 - lazy evaluation of values(值的缓式评估), 34
 - managing with RAII(用 RAII 管理), 139-142
 - modifying(修改), 34
 - with multiple addresses(具有多个地址).
 - 见 pointer comparison
 - with multiple types(具有多种类型). 见 polymorphism
 - polymorphic(多态), 3-5
 - structure and layout(结构和布局), 37-39
 - traits, 193-197
 - type constraints(类型约束), 111
 - virtual constructors(虚构造函数), 99-101
 - Offset(偏移量). 见 delta adjustment of class pointer;
 - pointers to, data members
 - Old-style cast operators, *versus* new style
 - (旧式转型操作符与新式风格), 29
 - operator delete
 - class-specific memory management
 - (特定于类的内存管理), 123-126
 - usual version(普通版本), 125
 - Operator function lookup(操作符函数查找), 91-92
 - operator new
 - class-specific memory management
 - (特定于类的内存管理), 123-126
 - versus* new operator(与 new 操作符), 119, 123, 143
 - placement new(定位 new), 119-121
 - usual version(普通版本), 125
 - Operator overloading(操作符重载). 另见 function overloading
 - auto_ptr, 147
 - constructors(构造函数), 143-144
 - exception safety(异常安全), 131
 - exceptions(异常), 143-144
 - function calls(函数调用), 91-92
 - function objects(函数对象), 63
 - infix calls(中缀调用), 90-92
 - new operator(新式操作符), 143-144
 - operator function lookup(操作符函数查找), 91-92
 - versus* overriding(与重写), 75
 - placement new(定位 new), 119
 - pointer arithmetic(指针算术), 149-151
 - policies(策略), 206
 - smart pointers(智能指针), 145-146
 - STL function objects(STL 函数对象), 72
 - STL (standard template library)(标准模板库), 11
 - Operator precedence, pointers to member functions(操作符
 - 优先级, 指向成员函数的指针), 58, 61
 - Orwell, George (veiled reference), 224
 - Overloading(重载)
 - > operator(-> 操作符), 145-146, 147-148
 - * (asterisk) operator(* 操作符), 145-146, 147-148
 - as communication, with other programmers
 - (与其他程序员交流), 214-215
 - function call operators(函数调用操作符), 63
 - function templates(函数模板), 213-215, 217-220
 - functions(函数). 见 function overloading
 - index operator(索引操作符), 35-36
 - operators(操作符). 见 operator overloading
 - operators, policies(操作符, 策略), 206
 - versus* overriding(与重写), 75-76
 - Overriding(重写)
 - functions, covariant return types
 - (函数, 协变返回类型), 107-109
 - versus* overloading(与重载), 75-76
- ## P
- Parameters, templates(参数, 模板), 153-154
 - Parentheses (())(括号)
 - function call operator(函数调用操作符), 58
 - grouping in declarations(声明中的分组), 61
 - grouping with pointer-to-member operators(用指向
 - 成员的操作符分组), 58
 - Partial specialization(局部特化), 161, 183-187, 197
 - Patterns(模式). 见 design patterns
 - Paul Revere and the Raiders, 143

Pointer arithmetic(指针算术),19,149-151

Pointer comparison(指针比较),97-98

Pointers(指针).另见 smart pointers

arrays of(数组指针),25

dereferencing(解引用),53-54

integers as(整数作为指针/指针不是整数),151

list iterators(list 迭代器),151

losing type information(丢失类型信息),98

managing buffers of(指针缓冲区的管理),25

multilevel(多级指针).见 pointers to, pointers

versus references(与引用),13

stacks of(指针的栈),185-187

subtracting(相减),151

Pointers to(指向……的指针)

array declarators(数组声明符),61-62

characters(字符),156

class members, versus pointers(指向类成员的指针,与指针),53-56

const(指向常量的指针)

versus const pointers(常量指针),21-23

converting to pointer to non-const(转换成指向非常量的指针),23

data members(数据成员),54-55

function declarators(函数声明符),61-62

functions(函数).见 callbacks; Command pattern;

function pointers

member functions(成员函数)

contravariance(逆变性),54-55

declaration syntax(声明语法),57-58

integrating with function objects(与函数对象整合),66

operator precedence(操作符优先级),58,61

versus pointers(与指针),57-59

simple pointer to function(指向函数的简单指针),58

virtualness(虚拟性),58

non-const, converting to pointer to const(非常量,转换成指向常量的指针),22-23

pointers(指针)

changing pointer values(改变指针值),26

conversions(转换),26-27

converting to pointer to const(转换成指向常量的指针),27

converting to pointer to non-const

(转换成指向非常量的指针),27

description(描述),25-27

managing buffers of pointers(指针缓冲区的管理),25-26

void,98

Policy(策略),205-208

Polymorphic base classes(多态基类),3-5

Polymorphism(多态),3-5

Predicates, STL function objects as(STL 函数对象作为判断式),73-74

Primary templates(主模板).另见 STL (standard template library); templates

explicit specialization(显式特化),155-158,183-187

instantiation(实例化),154

member specialization(成员特化),165

partial specialization(局部特化),161,183-187

SFINAE,218

specialization(特化),154

specializing for type information(针对类型信息的特化),183

Promulgation, conventions(发布,约定),193,197

Prototype,99-101

Q

QWAN (Quality Without A Name),90

R

RAII.另见 auto_ptr

Rebind convention for allocators(配置器的rebind约定),180

References(引用).另见 aliases

to arrays(数组引用),62

to const(常量引用),15

description(描述),13-15

to functions(函数引用),62

initialization(初始化),14-15

to non-const(非常量引用),15

null(空引用),13-14

versus pointers(与指针),13

register keyword(register关键字),233

reinterpret_cast operator(reinterpret_cast操作符),31

Resource acquisition is initialization(资源获取即初始化).见 RAII

Resource control(资源控制).见 RAII

RTTI (runtime type information)(运行期类型信息)

for capability query(能力查询),95

incorrect use of(RTTI 的不正确使用),103

runtime cost of(RTTI 的运行期开销),31

for safe down cast(安全的向下转型),31

S

Sakamoto, Kyu (veiled reference),42,68,70

Semaphores, resource control(信号量,资源控制).见 RAII

SFINAE (substitution failure is not an error)

(替换失败并非错误),217-220

Single-dimensional arrays, array formal arguments

(一维数组,数组形参),17-18

Smart pointers(智能指针).另见 function objects; pointers

versus auto_ptr(与 auto_ptr),148

list iterators(list 迭代器),11,151

operator overloading(操作符重载),145-146

templates(模板),145-146

Social commentary(社会舆论),7,10,71,190,195

Specialization(特化)

explicit(显式特化),183-187

partial(局部特化),183-187

SFINAE,218

templates(模板特化).见 templates, specialization

for type information(针对类型信息的特化),183

Specializing for type information(针对类型信息的特化),183

Standard template library (STL)(标准模板库).另见 primary

templates; templates

Static linkages, avoiding(避免静态连接),85

static_cast operator(static_cast 操作符),30-32

STL function objects(STL 函数对象)

as comparators(作为比较器),72-73

description(描述),71-74

as predicates(作为判断式),73-74

true/false questions((真/假问题),73-74

STL (standard template library)(标准模板库).另见 primary

templates; templates,11-12

Subcontractor, derived class as(派生类作为“转包者”),4-5

Substitution failure is not an error (SFINAE)

(替换失败并非错误),217-220

Swedish, and technical communication(瑞典,技术交流),99

T

Template argument deduction(模板实参推导),18,209-212

Template Method *versus* C++ templates(Template Method

与 C++ 模板),77

description(描述),77-79

Template template parameters(模板的模板参数),199-204

Templates(模板).另见 primary templates; STL

(standard template library)

arguments(实参)

customizing(定制).见 templates, specialization

description(描述),153-154

array formal arguments(数组形参),18-19

C++ *versus* Template Method(C++ 与 Template Method),77

customizing(定制).见 templates, specialization

disambiguation(消除歧义),169-172,179-181

ids(标识符),154

ignorance and(忽视),180

instantiation(实例化),154

member functions(成员函数),173-177

names(名字),154

nested names(嵌套名字),179

parameters(参数),153-154

smart pointers(智能指针),145-146

specialization(特化)

explicit(显式特化),155-159,183-187,209-212

partial(局部特化),161-164,183-187

terminology(术语),153-154

traits,197

traits,195-197

Terminology(术语)

assignment *versus* initialization(赋值和初始化),41-43

const pointers *versus* pointers to const(常量指针与
指向常量的指针),21-23

member templates(成员模板),174

new operator *versus* operator new(new 操作符与
operator new),119,123,143

new style cast operators *versus* old-style(新式转型操作
符与旧式风格),29

overloading *versus* overriding(重载与重写),75-76

pointers to class members *versus* pointers(类成员指针与
指针),53-56

pointers to const *versus* const pointers(指向常量的指针与
常量指针),21-23

pointers to member functions *versus* pointers(指向成员函数

的指针和指针),57-59

references *versus* pointers(引用与指针),13

template argument deduction(模板实参推导),209

Template Method *versus* C++ templates(Template Method 与 C++ 模板),77

templates(模板),153-154

wrappers(包装器),8

Traits

conventions(约定),193,197

description(描述),193-197

specialization(特化),197

templates(模板),195-197

Traits class(Traits 类),193-197

True/false questions(真/假问题),73-74

Type(类型)

container elements, determining(容器元素,确定),189-191

information, embedded(嵌入的类型信息),189-191

information about(类型信息),193-197

qualifiers, changing(改变类型修饰符),29-30,30-31

traits,193-197

Typename, disambiguation(利用 `typename` 消除歧义),169-172

U

Unnecessary static conventions(不必要的 `static` 约定),232

Unnecessary virtual conventions(不必要的 `virtual` 约定),232

User-defined types, assignment(用户自定义类型,赋值),41-43

Using declarations(using 声明),84

Using directives(using 指令),82-83

Usual operator new and operator delete(普通的 `operator new` 和 `operator delete`),125

V

Vandevorde, David,217-220

Variables, avoiding static linkage(变量,避免静态连接),85

Virtual constructors(虚构造函数),99-101

Virtual copy, assignment(虚复制,赋值),47

Virtual function pointers(虚函数指针),64-66

Virtual functions, class layout(虚函数,类的布局),37

Virtual inheritance, class layout(虚拟继承,类的布局),37-38

virtual keyword(`virtual` 关键字),231-233

volatile qualifiers, adding/removing(添加/移除 `volatile` 修饰符),29-30

W

What you see is what you get(所见即所得),37

Wilson, Flip (veiled reference),37

Z

Zoo animals, resource control(动物园里的动物,资源控制).见 RAII

代码示例索引

索引中的页码为英语原书页码，与书中页边标注的页码一致。

另请参考第 185 页“索引”。

A

- ABC class(ABC 类),113-114
- Abstract base class(抽象基类),113-114
 - ABC class(ABC 类),113-114
 - Action class(Action 类),69
 - Func class(Func 类),65
 - Rollable class(Rollable 类),93
- Access games(访问策略)
 - aFunc function(aFunc 函数),118
 - NoCopy class(NoCopy 类),111
 - NoHeap class(NoHeap 类),117-118
 - OnHeap class(OnHeap 类),118
- Action class(Action 类),69
- aFunc function(aFunc 函数),82-84,89,118
- Allocator(配置器)
 - AnAlloc class template(AnAlloc 类模板),179
 - AnAlloc::rebind member template
 - (AnAlloc::rebind 成员模板),179
 - SList class template(SList 类模板),180
- AnAlloc class template(AnAlloc 类模板),179
- AnAlloc::rebind member template
 - (AnAlloc::rebind 成员模板),179
- App class(App 类),78
- append function(append 函数),120-121
- App::startup member function(App::startup 成员函数),78
- Argument dependent lookup(实参相依的查找),89
- Array class template(Array 类模板),225
- Array<Circle, 7> explicit instantiation
 - (Array<Circle, 7> 显式实例化),227

- ArrayDeletePolicy class template
 - (ArrayDeletePolicy 类模板),207
- Array<T,n>::operator == template member function
 - (Array<T,n>::operator == 模板成员函数),226
- Assignment(赋值)
 - SList<T>::operator = member template
 - (SList<T>::operator = 成员模板),175
 - String::operator = member function
 - (String::operator = 成员函数),42,135
- aTemplateContext function template
 - (aTemplateContext 函数模板),132

B

- B class(B 类),59,75,87
- begForgiveness function(begForgiveness 函数),51
- Blob class(Blob 类),231
- Button class(Button 类),67,69
- Button::setAction member function
 - (Button::setAction 成员函数),136

C

- C class(C 类),54
- Callback(回调)
 - Action class(Action 类),69
 - begForgiveness function(begForgiveness 函数),51
- CanConvert class template(CanConvert 类模板),220
- Capability class,93
- cast function template(cast 函数模板),209
- CheckedPtr class template(CheckedPtr 类模板),145
- Circle class(Circle 类)
 - capability queries(能力查询),94
 - covariant return types(协变返回类型),107-108
 - pointers to class members(指向类成员的指针),55

- pointers to member functions(指向成员函数的指针),57
- CircleEditor class(CircleEditor类),108
- Class templates(类模板)
 - AnAlloc,179
 - Array,225
 - ArrayDeletePolicy,207
 - CanConvert,220
 - CheckedPtr,145
 - ContainerTraits,194
 - ContainerTraits< vector<T> >,197
 - ContainerTraits<const T *>,197
 - ContainerTraits<T *>,196
 - Heap,155,165
 - Heap<T *>,161
 - IsArray,187
 - IsClass,219
 - IsInt,183
 - IsPCM,187
 - IsPtr,184
 - MFunc,66
 - NoDeletePolicy,207
 - PFun1,215
 - PFun2,212,214
 - PtrCmp,162
 - PtrDeletePolicy,206-207
 - PtrList,169
 - PtrVector,25
 - ReadOnlySeq,191
 - SCollection,170
 - Seq,189-190
 - SList,173,180
 - Stack,185-186,200-202,205-206
 - Wrapper1,203
 - Wrapper2,203
 - Wrapper3,203-204
- Classes(类)
 - ABC,113-114
 - Action,69
 - App,78
 - B,59,75,87
 - Blob,231
 - Button,67,69
 - C,54
 - Circle
 - capability queries(能力查询),94
 - covariant return types(协变返回类型),107-108
 - pointers to class members(指向类成员的指针),55
 - pointers to member functions(指向成员函数的指针),57
 - CircleEditor,108
 - ContainerTraits<const char *>,196
 - ContainerTraits<ForeignContainer>,195
 - D,59,76,87
 - E,88
 - Employee,104-105
 - Fib,63
 - ForeignContainer,195
 - Func,65
 - Handle(句柄)
 - array allocation(数组分配),127-128
 - class-specific memory management(特定于类的内存管理),123-124
 - copy operations(复制操作),45
 - optional keywords(可选的关键字),232
 - restricting heap allocation(禁止或强制使用堆分配),118
 - Heap<char *>,157-158
 - Heap<const char *>,156
 - IsWarm,73
 - Meal,100
 - MyApp,78-79
 - MyContainer,170
 - MyHandle,124
 - NMFunc,66
 - NoCopy,111
 - NoHeap,117-118
 - ObservedBlob,97
 - OnHeap,118
 - PlayMusic,69
 - PopLess,72
 - rep,125
 - ResourceHandle,139
 - Rollable,93
 - S,38
 - Shape
 - capability queries(能力查询),93

- covariant return types(协变返回类型),107-108
- optional kwds(可选的关键字),231
- pointer comparison(指针比较),97
- pointers to class members(指向类成员的指针),55
- pointers to member functions(指向成员函数的指针),57
- ShapeEditor,108
- SharpBlob,231
- Spaghetti,100
- Square,94
- State,71,222-223
- String,41
- Subject,97
- T,39
- Temp,105
- Trace,141
- Wheel,94
- X,33-36,91
- cleanupBuf function(cleanupBuf函数),121
- Command pattern(Command 模式),69
- Comparator(比较器)
 - PopLess class(PopLess 类),72
 - popLess function(popLess 函数),71
 - PtrCmp class template(PtrCmp 类模板),162
 - strLess function(strLess 函数),157
- Computational constructor(计算性的构造函数),43
- ContainerTraits class template(ContainerTraits 类模板),194
- ContainerTraits< vector<T> > class template
 - (ContainerTraits< vector<T> > 类模板),197
- ContainerTraits<const char *> class
 - (ContainerTraits<const char *> 类),196
- ContainerTraits<const T *> class template
 - (ContainerTraits<const T *> 类模板),197
- ContainerTraits<ForeignContainer> class
 - (ContainerTraits<ForeignContainer> 类),195
- ContainerTraits<T *> class template
 - (ContainerTraits<T *> 类模板),196
- Copy assignment(复制赋值)
 - Handle::operator = member function
 - (Handle::operator = 成员函数),46-47
 - Handle::swap member function
 - (Handle::swap 成员函数),46

Covariant return(协变返回),107-108

D

D class(D 类),59,76,87

E

E class(E 类),88

Employee class(Employee 类),104-105

Exceptions(异常)

aTemplateContext function template

(aTemplateContext 函数模板),132

Button::setAction member function

{Button::setAction 成员函数},136

f function(f 函数),140-141

ResourceHandle class(ResourceHandle 类),139

String::operator = member function

(String::operator = 成员函数),135

Trace class(Trace 类),141

X::X member function(X::X 成员函数),133

Explicit instantiation(显式实例化)

Array<Circle, 7>,227

Heap<double>,167

Explicit specialization(显式特化)

ContainerTraits<const char *> class

(ContainerTraits<const char *> 类),196

ContainerTraits<ForeignContainer> class

(ContainerTraits<ForeignContainer> 类),195

Heap class template(Heap 类模板),155

Heap<char *> class(Heap<char *> 类),157-158

Heap<const char *> class

(Heap<const char *> 类),156

IsInt class template(IsInt 类模板),183

extractHeap function template

(extractHeap 函数模板),158

F

f function(f 函数),140-141

Factory Method

Circle class(Circle 类),108

Employee class(Employee 类),104-105

Shape class(Shape 类),108

Temp class(Temp 类),105
 Fib class(Fib 类),63
 fibonacci function (fibonacci 函数),64
 Fib::operator () member function
 (Fib::operator () 成员函数),63
 fill function template(fill 函数模板),170,172
 ForeignContainer class(ForeignContainer 类),195
 friend function(friend 函数),43
 Func class(Func 类),65
 Function object(函数对象)
 Action class(Action 类),69
 Fib class(Fib 类),63
 Func class(Func 类),65
 ISwarm class(ISwarm 类),73
 MFunc class template(MFunc 类模板),66
 NMFunc class(NMFunc 类),66
 PFun1 class template(PFun1 类模板),215
 PFun2 class template(PFun2 类模板),212,214
 PlayMusic class(PlayMusic 类),69
 PopLess class(PopLess 类),72
 PtrCmp class template(PtrCmp 类模板),162
 Function template overloading(函数模板重载)
 g function(g 函数),213
 g function template(g 函数模板),213
 makePFun function template(makePFun 函数模板),215
 Function templates(函数模板)
 aTemplateContext,132
 cast,209
 extractHeap,158
 fill,170,172
 g,213
 makePFun,212,214-215
 min,209
 process,19,189,191,194
 process_2d,19
 repeat,211
 set_2d,15
 slowSort,221-224
 swap,14,45
 zeroOut,211
 Functions(函数)
 aFunc,82-84,89,118

append,120-121
 begForgiveness,51
 cleanupBuf,121
 f,140-141
 fibonacci,64
 g,213
 genInfo,104
 integrate,65
 operator new,119
 org_semantics::operator +,82
 popLess,71
 scanTo,26
 someFunc,115
 String::operator +,43
 strLess,157
 swap,222

G

g function(g 函数),213
 g function template(g 函数模板),213
 Generic algorithms(泛型算法),221-224
 genInfo function(genInfo 函数),104

H

Handle class(Handle 类)
 array allocation(数组分配),127-128
 class-specific memory management(特定于类的
 内存管理),123-124
 copy operations(复制操作),45
 optional keywords(可选的关键字),232
 restricting heap allocation(禁止或强制使用堆分配),118
 Handle::operator = member function(Handle::opera
 tor = 成员函数),46-47
 Handle::operator delete member function(Handle::
 operator delete 成员函数),126
 Handle::operator new member function
 (Handle::operator new 成员函数),125
 Handle::swap member function(Handle::swap
 成员函数),46
 hasIterator preprocessor macro
 (hasIterator 预处理宏),219

Heap class template(Heap 类模板),155,165
 Heap<char *> class(Heap<char *> 类),157-158
 Heap<const char *> class
 (Heap<const char *> 类),156
 Heap<const char *>::pop member function
 (Heap<const char *>::pop 成员函数),166
 Heap<const char *>::push member function
 (Heap<const char *>::push 成员函数),157,166-167
 Heap<double> explicit instantiation (Heap<double>
 显式实例化),167
 Heap<T *> class template(Heap<T *> 类模板),161
 Heap<T *>::push template member function
 (Heap<T *>::push 模板成员函数),162
 Heap<T>::pop template member function
 (Heap<T>::pop 模板成员函数),156
 Heap<T>::push template member function
 (Heap<T>::push 模板成员函数),155
 Helper function(辅助函数),212,214

I

integrate function(integrate 函数),65
 Interface class(接口类)
 Action class(Action 类),69
 Func class(Func 类),65
 Rollable class(Rollable 类),93
 IsArray class template(IsArray 类模板),187
 IsClass class template(IsClass 类模板),219
 IsInt class template(IsInt 类模板),183
 IsPCM class template(IsPCM 类模板),187
 IsPtr class template(IsPtr 类模板),184
 is_ptr preprocessor macro(is_ptr 预处理宏),217
 ISwarm class(ISwarm 类),73

M

makePFun function template(makePFun 函数模板),212,214-215
 Meal class(Meal 类),100
 Member array new(成员 array new),118,128
 Member delete(成员删除),126
 Member functions(成员函数)
 App::startup,78
 Button::setAction,136
 Fib::operator ,63

Handle::operator =,46-47
 Handle::operator delete,126
 Handle::operator new,125
 Handle::swap,46
 Heap<const char *>::pop,166
 Heap<const char *>::push,157,166-167
 String::operator =,42,135
 String::String,42
 String::String,42-43
 X::getValue,34-35
 X::memFunc2,91
 X::X,133

Member new(成员 new)

Handle class(Handle 类),123-124
 Handle::operator new member function(Handle::
 operator new 成员函数),125
 MyHandle class(MyHandle 类),124

Member specialization(成员特化)

Heap<const char *>::pop member function
 (Heap<const char *>::pop 成员函数),166
 Heap<const char *>::push member function
 (Heap<const char *>::push 成员函数),157,166-167

Member templates(成员模板)

AnAlloc::rebind,179
 SList<T>::operator =,175
 SList<T>::SList,174-175
 SList<T>::sort,176

MFunc class template(MFunc 类模板),66

minimum function template(minimum 函数模板),209

Multiple inheritance(多重继承),97

MyApp class(MyApp 类),78-79

MyContainer class(MyContainer 类),170

MyHandle class(MyHandle 类),124

N

Namespaces(名字空间)

aFunc function(aFunc 函数),82-84
 org_semantics,81,89
 org_semantics namespace
 (org_semantics 名字空间),81
 org_semantics::operator + function
 (org_semantics::operator + 函数),82

NMFunc class(NMFunc 类),66
 NoCopy class(NoCopy 类),111
 NoDeletePolicy class template(NoDeletePolicy 类模板),207
 NoHeap class(NoHeap 类),117-118

O

ObservedBlob class(ObservedBlob 类),97
 OnHeap class(OnHeap 类),118
 operator new function(operator new 函数),119
 org_semantics namespace
 (org_semantics 名字空间),81,89
 org_semantics::operator + function
 (org_semantics::operator + 函数),82

P

Partial specialization(局部特化)

ContainerTraits< vector<T> > class template
 (ContainerTraits< vector<T> > 类模板),197
 ContainerTraits<const T * > class template
 (ContainerTraits<const T * > 类模板),197
 ContainerTraits<T * > class template
 (ContainerTraits<T * > 类模板),196
 Heap<T * > class template(Heap<T * > 类模板),161
 Heap<T * >::push template member function
 (Heap<T * >::push 模板成员函数),162
 IsArray class template(IsArray 类模板),187
 IsPCM class template(IsPCM 类模板),187
 IsPtr class template(IsPtr 类模板),184
 PFunc1 class template(PFunc1 类模板),215
 PFunc2 class template(PFunc2 类模板),212,214
 Placement new(定位new)
 append function(append 函数),120-121
 operator new function(operator new 函数),119
 PlayMusic class(PlayMusic 类),69
 POD,38
 Pointer arithmetic(指针算术)
 process_2d function template
 (process_2d 函数模板),19
 set_2d function template(set_2d 函数模板),15
 Policy(策略)
 ArrayDeletePolicy class template

 (ArrayDeletePolicy 类模板),207
 NoDeletePolicy class template
 (NoDeletePolicy 类模板),207
 PtrDeletePolicy class template
 (PtrDeletePolicy 类模板),206-207
 Stack class template(Stack 类模板),205-206
 PopLess class(PopLess 类),72
 popLess function(popLess 函数),71
 Predicate(判断式),73
 Preprocessor macro(预处理宏)
 hasIterator,219
 is_ptr,217
 process function template
 (process 函数模板),19,189,191,194
 process_2d function template(process_2d 函数模板),19
 Prototype
 Action class(Action 类),69
 Circle class(Circle 类),107
 Meal class(Meal 类),100
 PlayMusic class(PlayMusic 类),69
 Shape class(Shape 类),107
 Spaghetti class(Spaghetti 类),100
 PtrCmp class template(PtrCmp 类模板),162
 PtrDeletePolicy class template
 (PtrDeletePolicy 类模板),206-207
 PtrList class template(PtrList 类模板),169
 PtrVector class template(PtrVector 类模板),25

R

ReadonlySeq class template(ReadonlySeq 类模板),191
 rep class(rep 类),125
 repeat function template(repeat 函数模板),211
 ResourceHandle class(ResourceHandle 类),139
 Rollable class(Rollable 类),93

S

S class(S 类),38
 scanTo function(scanTo 函数),26
 SCollection class template(SCollection 类模板),170
 Seq class template(Seq 类模板),189-190
 set_2d function template(set_2d 函数模板),15

SFINAE

CanConvert class template(CanConvert 类模板),220

hasIterator preprocessor macro
(hasIterator 预处理宏),219

IsClass class template(IsClass 类模板),219

is_ptr preprocessor macro(is_ptr 预处理宏),217

Shape class(Shape 类)

- capability queries(能力查询),93
- covariant return types(协变返回类型),107-108
- optional kwds(可选的关键字),231
- pointer comparison(指针比较),97
- pointers to class members(指向类成员的指针),55
- pointers to member functions(指向成员函数的指针),57

ShapeEditor class(ShapeEditor 类),108

SharpBlob class(SharpBlob 类),231

SList class template(SList 类模板),173,180

SList<T>::empty template member function
(SList<T>::empty 模板成员函数),173

SList<T>::Node template member class
(SList<T>::Node 模板成员类),174

SList<T>::operator = member template
(SList<T>::operator = 成员模板),175

SList<T>::SList member template
(SList<T>::SList 成员模板),174-175

SList<T>::sort member template
(SList<T>::sort 成员模板),176

slowSort function template(slowSort 函数模板),221-224

smart pointer(智能指针),145

someFunc function(someFunc 函数),115

Spaghetti class(Spaghetti 类),100

Square class(Square 类),94

Stack class template(Stack 类模板),185-186,200-202,205-206

Stack<T>::push template member function
(Stack<T>::push 模板成员函数),185

State class(State 类),71,222-223

String class(String 类),41

String::operator + function
(String::operator + 函数),43

String::operator = member function
(String::operator = 成员函数),42,135

String::String member function
(String::String 成员函数),42,42-43

strLess function(strLess 函数),157

Subject class(Subject 类),97

swap function(swap 函数),222

swap function template(swap 函数模板),14,45

T

T class(T 类),39

Temp class(Temp 类),105

Template argument deduction(模板实参推导)

- cast function template(cast 函数模板),209
- makePFun function template(makePFun 函数模板),212,214
- minimum function template(minimum 函数模板),209
- repeat function template(repeat 函数模板),211
- zeroOut function template(zeroOut 函数模板),211

Template member class(模板成员类),174

Template member functions(模板成员函数)

- Array<T,n>::operator ==,226
- Heap<T *>::push,162
- Heap<T>::pop,156
- Heap<T>::push,155
- SList<T>::empty,173
- Stack<T>::push,185

Template Method

- App class(App 类),78
- App::startup member function(App 成员函数),78
- MyApp class(MyApp 类),78-79

Trace class(trace 类),141

Traits

ContainerTraits class template
(ContainerTraits 类模板),194

ContainerTraits< vector<T> > class template
(ContainerTraits< vector<T> > 类模板),197

ContainerTraits<const char *> class
(ContainerTraits<const char *> 类),196

ContainerTraits<const T *> class template
(ContainerTraits<const T *> 类模板),197

ContainerTraits<ForeignContainer> class
(ContainerTraits<ForeignContainer> 类),195

ContainerTraits<T *> class template
(ContainerTraits<T *> 类模板),196

W

Wheel class(Wheel 类),94

Wrapper1 class template(Wrapper1 类模板),203

Wrapper2 class template(Wrapper2 类模板),203

Wrapper3 class template(Wrapper3 类模板),203-204

X

X class(X 类),33-36,91

X::getValue member function

(X::getValue 成员函数),34-35

X::memFunc2 member function(X::memFunc2 成员函数),91

X::X member function(X::X 成员函数),133

Z

zeroOut function template(zeroOut 函数模板),211

W

Wheel class(Wheel 类),94

Wrapper1 class template(Wrapper1 类模板),203

Wrapper2 class template(Wrapper2 类模板),203

Wrapper3 class template(Wrapper3 类模板),203-204

X

X class(X 类),33-36,91

X::getValue member function

(X::getValue 成员函数),34-35

X::memFunc2 member function(X::memFunc2 成员函数),91

X::X member function(X::X 成员函数),133

Z

zeroOut function template(zeroOut 函数模板),211